Aprenda Computação com Python Documentation

Versão 1.1

Allen Downey, Jeff Elkner and Chris Meyers

12/08/2010

Conteúdo

Contents:

CAPÍTULO 1

Prefácio

Por Jeff Elkner

Este livro deve sua existência à colaboração possibilitada pela Internet e pelo movimento do software livre. Seus três autores – um professor universitário, um professor secundarista e um programador profissional – ainda não se encontraram pessoalmente, mas temos sido capazes de trabalhar em estreita colaboração e temos sido ajudados por muitos colegas maravilhosos que têm dedicado seu tempo e energia para ajudar a fazer deste um livro cada vez melhor.

Achamos que este livro é um testemunho dos benefícios e possibilidades futuras deste tipo de colaboração, cujo modelo tem sido posto em prática por Richard Stallman e pela Free Software Foundation.

1.1 Como e porque eu vim a usar Python

Em 1999, o Exame de Colocação Avançada em Ciência da Computação da Comissão de Faculdades (College Board's Advanced Placement (AP) Computer Science XXX) foi aplicado em C++ pela primeira vez. Como em muitas escolas secundárias através do país, a decisão de mudar linguagens teve um impacto direto no currículo de ciência da computação na Yorktown High School em Arlington, Virginia, onde leciono. Até então, Pascal era a linguagem didática para nossos cursos de primeiro ano e avançado. Mantendo a prática corrente de dar aos estudantes dois anos de exposição à mesma linguagem, tomamos a decisão de mudar para C++ no curso de primeiro ano para o ano letivo de 1997-98 de modo que estaríamos em sincronismo com a mudança da Comissão de Faculdades (College Board's XXX) em relação ao curso avançado (AP XXX) para o ano seguinte.

Dois anos depois, eu estava convencido que C++ foi uma escolha infeliz para introduzir os alunos em ciência da computação. Ao mesmo tempo em que é certamente uma linguagem de programação muito poderosa, também é uma linguagem extremamente difícil de aprender e de ensinar. Eu me encontrava constantemente lutando com a sintaxe difícil de C++ e as múltiplas maneiras de fazer a mesma coisa, e estava, como resultado, perdendo muitos alunos desnecessariamente. Convencido de que deveria existir uma linguagem melhor para a nossa classe de primeiro ano, fui procurar por uma alternativa a C++.

Eu precisava de uma linguagem que pudesse rodar nas máquinas em nosso laboratório Linux bem como nas plataformas Windows e Macintosh que a maioria dos alunos tinha em casa. Eu precisava que ela fosse gratuita e disponível eletronicamente, assim os alunos poderiam utilizá-la em casa independentemente de suas rendas. Eu queria uma linguagem que fosse utilizada por programadores profissionais, e que tivesse uma comunidade de desenvolvimento ativa em torno dela. Ela teria que suportar ambas, programação procedural e orientada a objetos. E, mais importante, deveria ser fácil de aprender e de ensinar. Quando considerei as alternativas tendo em mente aquelas metas, Python sobressaiu-se como a melhor candidata para a tarefa.

Pedi para um dos talentosos estudantes de Yorktown, Matt Ahrens, que experimentasse Python. Em dois meses ele não só aprendeu a linguagem como também escreveu uma aplicação chamada pyTicket que possibilitou à nossa equipe reportar problemas de tecnologia pela Web. Eu sabia que Matt não poderia ter finalizado uma aplicação daquele porte em período tão curto em C++, e esta realização, combinada com a avaliação positiva de Python dada por Matt, sugeriam que Python era a solução que eu estava procurando.

1.2 Encontrando um livro-texto

Tendo decidido usar Python em minhas aulas introdutórias de ciência da computação do ano seguinte, o problema mais urgente era a falta de um livro-texto disponível.

O conteúdo livre veio em socorro. Anteriormente naquele ano, Richard Stallman tinha me apresentado a Allen Downey. Ambos havíamos escrito a Richard expressando interesse em desenvolver conteúdo educacional livre. Allen já tinha escrito um livro-texto para o primeiro ano de ciência da computação, *How to Think Like a Computer Scientist*. Quando li este livro, soube imediatamente que queria utilizá-lo nas minhas aulas. Era o mais claro e proveitoso texto em ciência da computação que eu tinha visto. Ele enfatizava o processo de reflexão envolvido em programação em vez de características de uma linguagem em particular. Lê-lo fez de mim imediatamente um professor melhor.

O *How to Think Like a Computer Scientist* era não só um excelente livro, como também fora lançado sob uma licença pública GNU, o que significava que ele poderia ser usado livremente e modificado para atender as necessidades de seu usuário. Uma vez que eu havia decidido usar Python, me ocorreu que eu poderia traduzir a versão original do livro de Allen do Java para a nova linguagem. Apesar de não estar capacitado para escrever eu mesmo um livro-texto, tendo o livro de Allen para trabalhar, tornou possível para mim fazê-lo, ao mesmo tempo demonstrando que o modelo de desenvolvimento cooperativo tão bem utilizado em software poderia também funcionar para conteúdo educacional.

Trabalhar neste livro pelos últimos dois anos tem sido recompensador para mim e meus alunos, e eles tiveram um grande papel neste processo. A partir do momento em que eu podia fazer mudanças instantâneas assim que alguém encontrasse um erro ortográfico ou um trecho difícil, eu os encorajei a procurar por erros no livro, dando a eles pontos de bonificação cada vez que fizessem uma sugestão que resultasse em uma mudança no texto. Isto teve o duplo benefício de encorajá-los a ler o texto mais cuidadosamente e de ter o texto totalmente revisado por seus críticos mais importantes: alunos utilizando-o para aprender ciência da computação.

Para a segunda metade do livro, sobre programação orientada a objetos, eu sabia que seria preciso alguém com uma maior experiência do que a minha em programação real para fazê-lo corretamente. O livro esteve em estado inacabado por quase um ano até que a comunidade de software livre providenciasse mais uma vez os meios necessários para sua conclusão.

Eu recebi um e-mail de Chris Meyers mostrando interesse no livro. Chris é um programador profissional que começou a dar um curso de programação no ano anterior usando Python no Lane Community College em Eugene, Oregon. A perspectiva de dar aquele curso ligou Chris ao livro, e ele começou a ajudar o trabalho imediatamente. Até o final do ano letivo ele tinha criado um projeto colaborativo em nosso Website em `http://www.ibiblio.org/obp` chamado *Python for Fun* e estava trabalhando com alguns dos meus alunos mais avançados como um guru (master teacher XXX), guiando-os além de onde eu poderia levá-los.

1.3 Introduzindo programação com Python

O processo de traduzir e utilizar *How to Think Like a Computer Scientist* pelos últimos dois anos tem confirmado a conveniência de Python no ensino de alunos iniciantes. Python simplifica tremendamente os programas exemplo e torna idéias importantes de programação mais fáceis de ensinar.

O primeiro exemplo do texto ilustra este ponto. É o tradicional programa "Alô mundo", o qual na versão C++ do livro se parece com isto:

```
#include <iostream.h>

void main()
{
  cout << "Alô, mundo." << endl;
}
```

Na versão Python, ele se transforma em:

```
print "Alô, Mundo!"
```

Mesmo sendo um exemplo trivial, as vantagens do Python saltam aos olhos. O curso de Ciência da Computação I que ministro em Yorktown não tem pré-requisitos, assim, muitos dos alunos que veem esse exemplo estão olhando para o seu primeiro programa. Alguns deles estão indubitavelmente nervosos, por já terem ouvido falar que programação de computadores é difícil de aprender. A versão C++ tem sempre me forçado a escolher entre duas opções insatisfatórias: ou explicar os comandos `#include`, `void main()`, `{`, e `}` e arriscar confundir ou intimidar alguns dos alunos logo assim que iniciam, ou dizer a eles "Não se preocupem com todas estas coisas agora; falaremos sobre elas mais tarde," e correr o mesmo risco. O objetivo educacional neste ponto do curso é introduzir os alunos à idéia de comando em programação e vê-los escrever seu primeiro programa, deste modo introduzindo-os ao ambiente de programação. O programa em Python tem exatamente o que é necessário para conseguir isto, e nada mais.

Comparar o texto explicativo do programa em cada versão do livro ilustra ainda mais o que significa para o aluno iniciante. Existem treze parágrafos de explicação do "Alô, mundo!" na versão C++; na versão Python existem apenas dois. Mais importante, os onze parágrafos perdidos não se ocupam das "idéias chave" da programação de computadores, mas com a minúcia da sintaxe C++. Vejo a mesma coisa acontecendo através de todo o livro. Parágrafos inteiros simplesmente desaparecem da versão Python do texto porque a sintaxe muito mais clara de Python os torna desnecessários.

Utilizar uma linguagem de tão alto nível como Python, permite ao professor deixar para falar mais tarde sobre os níveis mais baixos, próximos à máquina, quando os alunos já terão a experiência necessária para ver com mais sentido os detalhes. Desta maneira podemos pedagogicamente "por em primeiro lugar as primeiras coisas". Um dos melhores exemplos disto é a maneira com que Python lida com variáveis. Em C++ uma variável é um nome para um lugar que guarda uma coisa. Variáveis têm de ser declaradas com seu tipo pelo menos em parte por que o tamanho do lugar a que se referem precisa ser predeterminado. Assim, a idéia de variável fica amarrada ao hardware da máquina. O conceito poderoso e fundamental de variável já é bastante difícil para o aluno iniciante (tanto em ciência da computação quanto em álgebra). Bytes e endereços não ajudam neste caso. Em Python uma variável é um nome que se refere a uma coisa. Este é um conceito muito mais intuitivo para alunos iniciantes e está muito mais próximo do significado de "variável" que eles aprenderam em seus cursos de matemática. Eu tive muito menos dificuldade em ensinar variáveis este ano do que tive no passado, e gastei menos tempo ajudando aos alunos com problemas no uso delas.

Um outro exemplo de como Python ajuda no ensino e aprendizagem de programação é em sua sintaxe para função. Meus alunos têm sempre tido grande dificuldade na compreensão de funções. O problema principal gira em torno da diferença entre a definição de uma função e a chamada de uma função, e a distinção relacionada entre um parâmetro e um argumento. Python vem em auxílio com uma sintaxe não apenas curta quanto bela. As definições de função começam com `def`, então eu simplesmente digo aos meus alunos "Quando você define uma função, comece com `def`, seguido do nome da função que você está definindo; quando você chama uma função, simplesmente chame-a digitando o nome dela". Parâmetros ficam nas definições; argumentos vão com as chamadas. Não existem tipos de retorno, tipos de parâmetro ou passagem de parâmetros por valor ou por referência no meio do caminho, permitindo-me ensinar funções em menos da metade do tempo que isto me tomava anteriormente, com uma melhor compreensão.

A utilização do Python tem melhorado a efetividade de nosso programa em ciência da computação para todos os estudantes. Eu vejo um nível geral de sucesso muito mais alto e um nível mais baixo de frustração do que experimentei durante os dois anos em que ensinei C++. Eu avanço mais rápido com melhores resultados. Mais alunos deixam o curso com a habilidade de criar programas significativos e com uma atitude positiva em relação a experiência de programação que isso traz.

1.4 Construindo uma comunidade

Tenho recebido e-mails de todo o planeta de pessoas utilizando este livro para aprender ou ensinar programação. Uma comunidade de usuários tem começado a emergir e muitas pessoas têm contribuído com o projeto enviando seus materiais para o Website cooperativo em:

`http://www.thinkpython.com`

Com a publicação do livro em formato impresso, minha expectativa quanto ao crescimento da comunidade de usuários é que ela seja contínua e acelerada. O surgimento desta comunidade de usuários e a possibilidade que sugere de colaboração semelhante entre educadores tem sido para mim a parte mais excitante do trabalho neste projeto. Trabalhando juntos, podemos aumentar a qualidade do material disponível para o nosso uso e poupar tempo valioso. Eu convido você a se juntar a nossa comunidade e espero ouvir algo de você. Por favor, escreva para os autores em `feedback@thinkpython.com`.

Jeffrey Elkner

Yorktown High School

Arlington, Virginia

CAPÍTULO 2

Apresentação

Por David Beazley

Como educador, pesquisador e autor de livros, regozija-me ver completo este trabalho. Python é uma linguagem de programação divertida e extremamente fácil de usar que tem ganho forte popularidade nestes últimos poucos anos. Desenvolvida dez anos atrás por Guido van Rossun, a sintaxe simples do Python e seu sentido geral são grandemente derivados do ABC, uma linguagem didática que foi desenvolvida nos anos 80. Entretanto, Python também foi criado para solucionar problemas reais e tomou emprestado uma grande quantidade de características de linguagens de programação como C++, Java, Modula-3 e Scheme. Por causa disso, uma das mais notáveis características do Python é o grande apelo que tem junto a desenvolvedores profissionais de software, cientistas, pesquisadores, artistas e educadores.

A Despeito deste apelo do Python junto às mais variadas comunidades, você pode ainda estar pensando ?por que Python?? ou ?por que ensinar programação com Python??. Responder à estas perguntas não é uma tarefa fácil ? especialmente se a opinião pública está do lado de alternativas mais masoquistas como C++ e Java. Entretanto, eu acho que a resposta mais direta é que programar com Python é um bocado divertido e mais produtivo.

Quando ministro cursos de ciências da computação, o que desejo é cobrir conceitos importantes além de tornar a matéria interessante e os alunos participativos. Infelizmente, existe uma tendência entre os cursos introdutórios de programação a focar atenção demais em abstrações matemáticas, e de frustração entre os alunos com problemas enfadonhos e inoportunos relacionados a detalhes de sintaxe em baixo nível, compilação e a imposição de regras que aparentemente só um *expert* pode compreender (enforcement of seemingly arcane rules XXX). Embora alguma abstração e formalismo sejam importantes para engenheiros profissionais de software e estudantes que planejam continuar seus estudos em ciências da computação, escolher tal abordagem em um curso introdutório faz da ciência da computação algo entediante. Quando ministro um curso, não desejo uma sala cheia de alunos sem inspiração. Em vez disso, preferiria muito mais vê-los tentando solucionar problemas interessantes explorando idéias diferentes, trilhando caminhos não convencionais, quebrando regras, e aprendendo a partir de seus erros. Fazendo assim, não pretendo desperdiçar metade de um semestre tentando explicar problemas obscuros de sintaxe, mensagens ininteligíveis de compiladores ou as várias centenas de maneiras pelas quais um programa pode gerar uma falha geral de proteção.

Uma das razões pelas quais eu gosto de Python é que ele oferece um equilíbrio realmente bom entre o lado prático e o lado conceitual. Sendo Python interpretado, os iniciantes podem pegar a linguagem e começar a fazer coisas legais quase imediatamente sem se perderem em problemas de compilação e ligação (linking XXX). Além disso, Python vem com uma grande biblioteca de módulos que podem ser utilizados para fazer todo tipo de tarefa, desde a programação para a web até gráficos. Com tal enfoque prático temos uma bela maneira de alcançar o engajamento dos alunos e permitir que eles finalizem projetos significativos. Entretanto, Python também pode servir de excelente embasamento para a introdução de conceitos importantes em ciência da computação. Já que Python suporta plenamente procedimentos (*procedures*) e classes, os alunos podem ser gradualmente introduzidos a tópicos como abstração procedural, estruturas de dados, e programação orientada a objetos ? todos aplicáveis em cursos posteriores de Java ou C++.

Python ainda toma emprestado certas características de linguagens de programação funcionais e pode ser usado para introduzir conceitos cujos detalhes poderiam ser aprofundados em cursos de Scheme e Lisp.

Lendo o prefácio de Jeffrey, fiquei impressionado com seu comentário de que Python o fez ver um ?maior nível de sucesso e um menor nível de frustração? o que lhe permitiu ?progredir mais depressa com resultados melhores?. Embora estes comentários refiram-se aos seus cursos introdutórios, eu às vezes uso Python exatamente pelas mesmas razões em cursos avançados de pós-graduação (graduate = pos-graduacao XXX) em ciência da computação na Universidade de Chicago. Nestes cursos, enfrento constantemente a assustadora tarefa de cobrir muitos tópicos difíceis em um rapidíssimo trimestre (quarter XXX) de nove semanas. Embora me seja possível inflingir um bocado de dor e sofrimento pelo uso de uma linguagem como C++, tenho percebido muitas vezes que este enfoque é contraproducente ? especialmente quando o curso é sobre um tópico não relacionado apenas com ?programar?. Acho que usar Python me permite um melhor foco no tópico em questão, enquanto permite que os alunos completem projetos substanciais em classe.

Embora Python seja ainda uma linguagem jovem e em evolução, acredito que tem um futuro brilhante em educação. Este livro é um passo importante nessa direção.

David Beazley

Universidade de Chicago

Autor de *Python Essencial Reference*

CAPÍTULO 3

Capítulo 1: O caminho do programa

Tópicos

O objetivo deste livro é ensinar o leitor a pensar como um cientista da computação. Essa maneira de pensar combina algumas das melhores características da matemática, da engenharia e das ciências naturais. Como os matemáticos, os cientistas da computação usam linguagens formais para representar ideias (especificamente, computações). Como os engenheiros, eles projetam coisas, montando sistemas a partir de componentes e avaliando as vantagens e desvantagens de diferentes alternativas. Como os cientistas naturais, eles observam o comportamento de sistemas complexos, formulam hipóteses e testam previsões.

A habilidade mais importante de um cientista da computação é a **solução de problemas**. Solução de problemas é a habilidade de formular questões, pensar criativamente sobre soluções possíveis e expressar uma solução de forma clara e precisa. Ocorre que aprender a programar é uma excelente oportunidade de praticar a habilidade da solução de problemas. É por isso que este capítulo se chama "O caminho do programa".

Em certo nível, você estará aprendendo a programar, habilidade que é útil em si mesma. Em outro nível, você usará a programação como um meio para atingir um objetivo. À medida que você for avançando na leitura, esse objetivo ficará mais claro.

3.1 1.1 A linguagem de programação Python

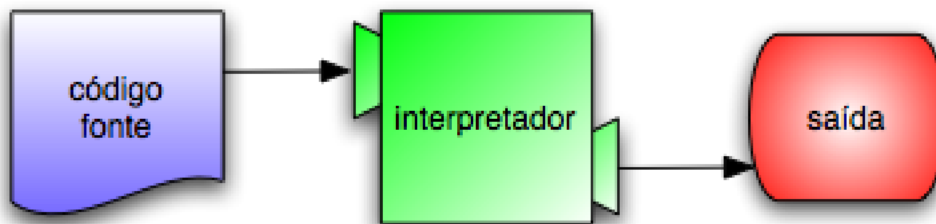
Python é a linguagem de programação que você vai estudar neste livro. Python é um exemplo de **linguagem de programação de alto nível**. Outras linguagens de alto nível de que você já pode ter ouvido falar são C++, PHP e Java.

Como você pode deduzir a partir da expressão "linguagem de alto nível", também existem as "linguagens de baixo nível", às vezes chamadas de "linguagens de máquina" ou "linguagem assembly" (linguagens de montagem). De forma simples, o computador só consegue executar programas escritos em linguagens de baixo nível. Deste modo, programas escritos em linguagens de alto nível precisam ser processados antes que possam rodar. Esse processamento extra toma algum tempo, o que é uma pequena desvantagem em relação às linguagens de alto nível.

Mas as vantagens são enormes. Primeiro, é muito mais fácil programar em uma linguagem de alto nível. É mais rápido escrever programas em uma linguagem de alto nível; eles são mais curtos e mais fáceis de ler, e há maior probabilidade de esterem corretos. Segundo, as linguagens de alto nível são **portáveis**, o que significa que podem rodar em diferentes tipos de computador, com pouca ou nenhuma modificação. Programas em baixo nível só podem rodar em um único tipo de computador e precisam ser re-escritos para rodar em outro tipo.

Devido a essas vantagens, quase todos os programas são escritos em linguagens de alto nível. As de baixo nível são utilizadas somente para umas poucas aplicações especializadas.

Dois tipos de programas processam linguagens de alto nível, traduzindo-as em linguagens de baixo nível: **interpretadores** e **compiladores**. O interpretador lê um programa escrito em linguagem de alto nível e o executa, ou seja, faz o que o programa diz. Ele processa o programa um pouco de cada vez, alternadamente: hora lendo algumas linhas, hora executando essas linhas e realizando cálculos.



O compilador lê o programa e o traduz completamente antes que o programa comece a rodar. Neste caso, o programa escrito em linguagem de alto nível é chamado de **código fonte**, e o programa traduzido é chamado de **código objeto** ou **executável**. Uma vez que um programa é compilado, você pode executá-lo repetidamente, sem que precise de nova tradução.



Python é considerada uma linguagem interpretada, pois os programas em Python são executados por um interpretador. Existem duas maneiras de usar o interpretador: no modo de linha de comando e no modo de script. No modo de linha de comando, você digita programas em Python e o interpretador mostra o resultado:

```
$ python
Python 2.5.2 (r252:60911, Jan  4 2009, 17:40:26)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

A primeira linha deste exemplo é o comando que inicia o interpretador Python. As três linhas seguintes são mensagens do interpretador. A quarta linha começa com `>>>`, que é o sinal usado pelo interpretador para indicar que ele está pronto. No exemplo anterior, digitamos `print 1 + 1` e o interpretador respondeu `2`.

Você também pode escrever um programa em um arquivo e usar o interpretador para executar o conteúdo desse arquivo.

Um arquivo como este é chamado de **script**. Por exemplo, usamos um editor de texto para criar um arquivo chamado `leticia.py` com o seguinte conteúdo:

```
print 1 + 1
```

Por convenção, arquivos que contenham programas em Python têm nomes que terminam com `.py`.

Para executar o programa, temos de dizer ao interpretador o nome do script:

```
$ python leticia.py
2
```

Em outros ambientes de desenvolvimento, os detalhes da execução de programas podem ser diferentes. Além disso, a maioria dos programas são mais interessantes do que esse.

A maioria dos exemplos neste livro são executados a partir da linha de comando. Trabalhar com a linha de comando é conveniente no desenvolvimento e testagem de programas, porque você pode digitar os programas e executá-los imediatamente. Uma vez que você tem um programa que funciona, deve guardá-lo em um script, de forma a poder executá-lo ou modificá-lo no futuro.

3.2 1.2 O que é um programa?

Um **programa** é uma sequência de instruções que especificam como executar um cálculo ou determinada tarefa. Tal tarefa pode matemática, como solucionar um sistema de equações ou encontrar as raízes de um polinômio, mas também pode ser simbólica, como buscar e substituir uma palavra em um documento ou (estranhamente) compilar um programa.

Os detalhes são diferentes em diferentes linguagens, mas algumas instruções básicas aparecem em praticamente todas as linguagens:

entrar: Pegar dados do teclado, de um arquivo ou de algum outro dispositivo de entrada.

sair: Mostrar dados na tela ou enviar dados para um arquivo ou outro dispositivo de saída.

calcular: Executar operações matemáticas básicas, como adição e multiplicação.

executar condicionalmente: Checar certas condições e executar a sequência apropriada de instruções.

repetir: Executar alguma ação repetidamente, normalmente com alguma variação.

Acredite se quiser: isso é praticamente tudo. Todos os programas que você já usou, não importa quão complicados, são feitos de instruções mais ou menos parecidas com essas. Assim, poderíamos definir programação como o processo de dividir uma tarefa grande e complexa em subtarefas cada vez menores, até que as subtarefas sejam simples o suficiente para serem executadas com uma dessas instruções básicas.

Isso pode parecer um pouco vago, mas vamos voltar a esse tópico mais adiante, quando falarmos sobre **algoritmos**.

3.3 1.3 O que é depuração (*debugging*)?

Programar é um processo complicado e, como é feito por seres humanos, frequentemente conduz a erros. Por mero capricho, erros em programas são chamados de **bugs** e o processo de encontrá-los e corrigi-los é chamado de **depuração** (*debugging*).

Três tipos de erro podem acontecer em um programa: erros de sintaxe, erros em tempo de execução (*runtime errors*) e erros de semântica (também chamados de erros de lógica). Distinguir os três tipos ajuda a localizá-los mais rápido:

3.3.1 1.3.1 Erros de sintaxe

O interpretador do Python só executa um programa se ele estiver sintaticamente correto; caso contrário, o processo falha e retorna uma mensagem de erro. **Sintaxe** se refere à estrutura de um programa e às regras sobre esta estrutura. Por exemplo, em português, uma frase deve começar com uma letra maiúscula e terminar com um ponto.

esta frase contém um **erro de sintaxe**. Assim como esta

Para a maioria dos leitores, uns errinhos de sintaxe não chegam a ser um problema significativo e é por isso que conseguimos ler a poesia moderna de e. e. cummings sem cuspir mensagens de erro. Python não é tão indulgente. Se o seu programa tiver um único erro de sintaxe em algum lugar, o interpretador Python vai exibir uma mensagem de erro e vai terminar - e o programa não vai rodar. Durante as primeiras semanas da sua carreira como programador, você provavelmente perderá um bocado de tempo procurando erros de sintaxe. Conforme for ganhando experiência, entretanto, cometerá menos erros e os localizará mais rápido.

3.3.2 1.3.2 Erros em tempo de execução (*runtime errors*)

O segundo tipo de erro é o erro de *runtime*, ou erro em tempo de execução, assim chamado porque só aparece quando você roda o programa. Esses erros são também conhecidos como **exceções**, porque normalmente indicam que alguma coisa excepcional (e ruim) aconteceu.

Erros de runtime são raros nos programas simples que você vai ver nos primeiros capítulos - então, vai demorar um pouco até você se deparar com um erro desse tipo.

3.3.3 1.3.3 Erros de semântica (ou de lógica)

O terceiro tipo de erro é o **erro de semântica** (mais comumente chamado erro de lógica). Mesmo que o seu programa tenha um erro de semântica, ele vai rodar com sucesso, no sentido de que o computador não vai gerar nenhuma mensagem de erro. Só que o programa não vai fazer a coisa certa, vai fazer alguma outra coisa. Especificamente, aquilo que você tiver dito para ele fazer (o computador trabalha assim: seguindo ordens).

O problema é que o programa que você escreveu não é aquele que você queria escrever. O significado do programa (sua semântica ou lógica) está errado. Identificar erros semânticos pode ser complicado, porque requer que você trabalhe de trás para frente, olhando a saída do programa e tentando imaginar o que ele está fazendo.

3.3.4 1.3.4 Depuração experimental (*debugging*)

Uma das habilidades mais importantes que você vai adquirir é a de depurar. Embora possa ser frustrante, depurar é uma das partes intelectualmente mais ricas, desafiadoras e interessantes da programação.

De certa maneira, a depuração é como um trabalho de detetive. Você se depara com pistas, e tem que deduzir os processos e eventos que levaram aos resultados que aparecem.

Depurar também é como uma ciência experimental. Uma vez que você tem uma ideia do que está errado, você modifica o seu programa e tenta de novo. Se a sua hipótese estava correta, então você consegue prever o resultado da modificação e fica um passo mais perto de um programa que funciona. Se a sua hipótese estava errada, você tem que tentar uma nova. Como Sherlock Holmes mostrou: "Quando você tiver eliminado o impossível, aquilo que restou, ainda que improvável, deve ser a verdade." (Arthur Conan Doyle, *O signo dos quatro*).

Para algumas pessoas, programação e depuração são a mesma coisa. Ou seja, programar é o processo de gradualmente depurar um programa, até que ele faça o que você quer. A ideia é começar com um programa que faça *alguma coisa* e ir fazendo pequenas modificações, depurando-as conforme avança, de modo que você tenha sempre um programa que funciona.

Por exemplo, o Linux é um sistema operacional que contém milhares de linhas de código, mas começou como um programa simples, que Linus Torvalds usou para explorar o chip Intel 80386. De acordo com Larry Greenfield, "Um dos primeiros projetos de Linus Torvalds foi um programa que deveria alternar entre imprimir AAAA e BBBB. Isso depois evoluiu até o Linux". (*The Linux User's Guide* Versão Beta 1)

Capítulos posteriores farão mais sugestões sobre depuração e outras práticas de programação.

3.4 1.4 Linguagens naturais e linguagens formais

Linguagens naturais são as linguagens que as pessoas falam, como o português, o inglês e o espanhol. Elas não foram projetadas pelas pessoas (muito embora as pessoas tentem colocar alguma ordem nelas); elas evoluíram naturalmente.

Linguagens formais são linguagens que foram projetadas por pessoas, para aplicações específicas. Por exemplo, a notação que os matemáticos usam é uma linguagem formal, que é particularmente boa em denotar relações entre números e símbolos. Os químicos usam uma linguagem formal para representar a estrutura química das moléculas. E, mais importante:

> **Linguagens de programação são linguagens formais que foram desenvolvidas para expressar computações.**

As linguagens formais tendem a ter regras estritas quanto à sintaxe. Por exemplo, **3 + 3 = 6** é uma expressão matemática sintaticamente correta, mas **3=+6$** não é. **H2O** é um nome químico sintaticamente correto, mas **2Zz** não é.

As regras de sintaxe são de dois tipos, um relacionado aos **símbolos**, outro à estrutura. Os símbolos são os elementos básicos da linguagem, como as palavras, números, e elementos químicos. Um dos problemas com **3=+6$** é que **$** não é um símbolo válido em linguagem matemática (pelo menos até onde sabemos). Do mesmo modo, **2Zz** é inválida porque não existe nenhum elemento cuja abreviatura seja **Zz**.

O segundo tipo de erro de sintaxe está relacionado à estrutura de uma expressão, quer dizer, ao modo como os símbolos estão arrumados. A expressão **3=+6$** é estruturalmente inválida, porque você não pode colocar um sinal de "mais" imediatamente após um sinal de "igual". Do mesmo modo, fórmulas moleculares devem ter índices subscritos colocados depois do nome do elemento, não antes.

> Faça este exercício: crie o que pareça ser uma frase bem estruturada em português com símbolos irreconhecíveis dentro dela. Depois escreva outra frase com todos os símbolos válidos, mas com uma estrutura inválida.

Quando você lê uma frase em português ou uma expressão em uma linguagem formal, você tem de imaginar como é a estrutura da frase (embora, em uma linguagem natural, você faça isso inconscientemente). Este processo, na computação, é chamado **parsing** (análise sintática).

Por exemplo, quando você ouve a frase, "Caiu a ficha", entende que "a ficha" é o sujeito e "caiu" é o verbo. Uma vez que você analisou a frase, consegue entender o seu significado, ou a semântica da frase. Assumindo que você saiba o que é uma ficha e o que significa cair, você entenderá o sentido geral dessa frase.

Muito embora as linguagens formais e as naturais tenham muitas características em comum (símbolos, estrutura, sintaxe e semântica), existem muitas diferenças:

ambiguidade: As linguagens naturais estão cheias de ambiguidades, que as pessoas contornam usando pistas contextuais e outras informações. Já as linguagens formais são desenvolvidas para serem quase ou totalmente desprovidas de ambiguidade, o que significa que qualquer expressão tem precisamente só um sentido, independentemente do contexto.

redundância: Para compensar a ambiguidade e reduzir maus entendidos, emprega-se muita redundância nas linguagens naturais, o que frequentemente as torna prolixas. As linguagens formais são menos redundantes e mais concisas.

literalidade: As linguagens naturais estão cheias de expressões idiomáticas e metáforas. Se eu digo "Caiu a ficha", é possível que não exista ficha nenhuma, nem nada que tenha caído. Nas linguagens formais, não há sentido ambíguo.

Pessoas que crescem falando uma linguagem natural, ou seja, todo mundo, muitas vezes têm dificuldade de se acostumar com uma linguagem formal. De certa maneira, a diferença entre linguagens formais e naturais é como a diferença entre poesia e prosa, porém mais acentuada:

poesia: As palavras são usadas pela sua sonoridade, além de seus sentidos, e o poema como um todo cria um efeito ou uma reação emocional. A ambiguidade não é apenas frequente, mas na maioria das vezes, proposital.

prosa: O sentido literal das palavras é mais importante, e a estrutura contribui mais para o significado. A prosa é mais fácil de analisar do que a poesia, mas ainda é, muitas vezes, ambígua.

programas: O significado de um programa de computador é exato e literal, e pode ser inteiramente entendido pela análise de seus símbolos e de sua estrutura.

Aqui vão algumas sugestões para a leitura de programas (e de outras linguagens formais). Primeiro, lembre-se de que linguagens formais são muito mais densas do que linguagens naturais, por isso, é mais demorado lê-las. A estrutura também é muito importante, logo, geralmente não é uma boa ideia ler de cima para baixo, da esquerda para a direita. Em vez disso, aprenda a analisar o programa na sua cabeça, identificando os símbolos e interpretando a estrutura. Finalmente, os detalhes são importantes. Pequenas coisas, como erros ortográficos e má pontuação, com as quais você pode se safar nas linguagens naturais, podem fazer uma grande diferença em uma linguagem formal.

3.5 1.5 O primeiro programa

Tradicionalmente, o primeiro programa escrito em uma nova linguagem de programação é chamado de "Alô, Mundo!" porque tudo que ele faz é apresentar as palavras "Alô, Mundo!". Em Python, ele é assim:

```
print "Alô, Mundo!"
```

Isso é um exemplo de um **comando print**, que, na realidade, não "imprime" nada em papel. Ele apresenta o valor na tela. Neste caso, o resultado são as palavras:

```
Alô, Mundo!
```

As aspas no programa marcam o começo e o fim do valor, elas não aparecem no resultado final.

Algumas pessoas julgam a qualidade de uma linguagem de programação pela simplicidade do programa "Alô, Mundo!". Por esse padrão, Python se sai tão bem quanto possível.

3.6 1.6 Glossário

algoritmo (*algorithm*) Processo geral para solução de uma certa categoria de problema.

análise sintática (*parse*) Examinar um programa e analisar sua estrutura sintática.

bug Erro em um programa.

código fonte (*source code*) Um programa em uma linguagem de alto nível, antes de ter sido compilado.

código objeto (*object code*) A saída do compilador, depois que ele traduziu o programa.

comando *print* (*'print' statement*) Instrução que leva o interpretador Python a apresentar um valor na tela.

compilar (*compile*) Traduzir todo um programa escrito em uma linguagem de alto nível para uma de baixo nível de uma só vez, em preparação para uma execução posterior.

depuração (*debugging*) O processo de encontrar e remover qualquer um dos três tipos de erros de programação.

erro de semântica ou lógica (*semantic error*) Erro em um programa, que o leva a fazer algo diferente do que pretendia o programador.

erro de sintaxe (*syntax error*) Erro em um programa, que torna impossível a análise sintática (logo, também impossível a interpretação).

erro em tempo de execução (*runtime error*) Erro que não ocorre até que o programa seja executado, mas que impede que o programa continue.

exceção (*exception*) Um outro nome para um erro em tempo de execução ou erro de *runtime*.

executável (*executable*) Um outro nome para código objeto que está pronto para ser executado.

interpretar (*interpret*) Executar um programa escrito em uma linguagem de alto nível, traduzindo-o uma linha de cada vez.

linguagem de alto nível (*high-level language*) Uma linguagem de programação como Python: projetada para ser fácil para os seres humanos utilizarem.

linguagem de baixo nível (*low-level language*) Uma linguagem de programação que é concebida para ser fácil para um computador, tal como a linguagem de máquina ou a linguagem montagem (*assembly language*)

linguagem formal (*formal language*) Qualquer linguagem desenvolvida pelas pessoas para propósitos específicos, tais como, a representação de ideias matemáticas ou programas de computadores; todas as linguagens de programação são linguagens formais.

linguagem natural (*natural language*) Qualquer língua falada pelos seres humanos que tenha evoluído naturalmente.

portabilidade (*portability*) Propriedade que um programa tem de rodar em mais de um tipo de computador.

programa (*program*) Conjunto de instruções que especifica uma computação.

script Um programa guardado em um arquivo (normalmente um que será interpretado).

semântica (*semantics*) O significado de um programa.

símbolo (*token*) Um elemento básico da estrutura sintática de um programa, análogo a uma palavra em uma linguagem natural.

sintaxe (*syntax*) A estrutura de um programa.

solução de problemas (*problem solving*) O processo de formular um problema, encontrar uma solução e expressar esta solução.

Capítulo 2: Variáveis, expressões e comandos

Tópicos

4.1 2.1 Valores e tipos

O **valor** (por exemplo, letras e números) é uma das coisas fundamentais que um programa manipula. Os valores que já vimos até agora foram o `2` (como resultado, quando adicionamos `1 + 1`) e `"Alô, Mundo!"`.

Esses valores pertencem a **tipos** diferentes: `2` é um inteiro, e `"Alô, Mundo!"` é uma **string**, assim chamada porque "string", em inglês, quer dizer sequência, série, cadeia (de caracteres), ou neste caso, "série de letras". Você (e o interpretador) consegue identificar strings porque elas aparecem entre aspas.

O comando print também funciona com inteiros:

```
>>> print 4
4
```

Se você estiver em dúvida sobre qual é o tipo de um determinado valor, o interpretador pode revelar:

```
>>> type("Alô, Mundo!")
<type 'string'>
```

```
>>> type(17)
<type 'int'>
```

Nenhuma surpresa: strings pertencem ao tipo `string` e inteiros pertencem ao tipo `int`. Menos obviamente, números com um ponto decimal pertencem a um tipo chamado `float`, porque estes números são representados em um formato chamado **ponto flutuante** [1]:

```
>>> type(3.2)
<type 'float'>
```

O que dizer de valores como `"17"` e `"3.2"`? Eles parecem números, mas estão entre aspas, como strings:

```
>>> type("17")
<type 'string'>
>>> type("3.2")
<type 'string'>
```

Eles são strings.

Ao digitar um número grande, é tentador usar pontos entre grupos de três dígitos, assim: 1.000.000. Isso não funciona por que Python usa o ponto como separador decimal. Usar a vírgula, como se faz em inglês, resulta numa expressão válida, mas não no número que queríamos representar:

```
>>> print 1,000,000
1 0 0
```

Não é nada do que se esperava! Python interpreta `1,000,000` como uma tupla, algo que veremos no Capítulo 9. Por hora, lembre-se apenas de não colocar vírgulas nos números.

4.2 2.2 Variáveis

Uma das características mais poderosas de uma linguagem de programação é a habilidade de manipular **variáveis**. Uma variável é um nome que se refere a um valor.

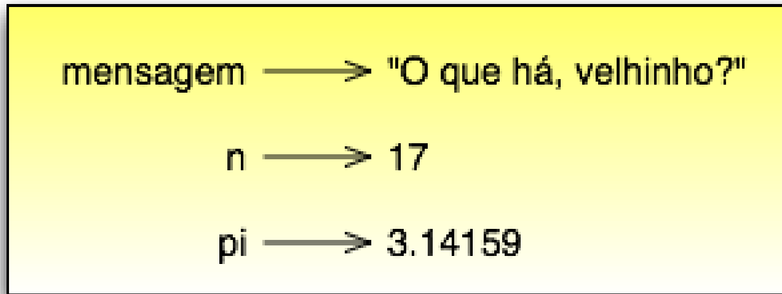
O **comando de atribuição** cria novas variáveis e dá a elas valores:

```
>>> mensagem = "E aí, Doutor?"
>>> n = 17
>>> pi = 3.14159
```

Este exemplo faz três atribuições. A primeira atribui a string `"E aí, Doutor?"` a uma nova variável chamada `mensagem`. A segunda dá o valor inteiro `17` a `n`, e a terceira atribui o número de ponto flutuante `3.14159` à variável chamada `pi`.

Uma maneira comum de representar variáveis no papel é escrever o nome delas com uma seta apontando para o valor da variável. Esse tipo de figura é chamado de **diagrama de estado** porque mostra em que estado cada variável está (pense nisso como o estado de espírito da variável). O diagrama a seguir mostra o resultado das instruções de atribuição:

[1] N.T.: Observe o uso de ponto no lugar da vírgula para separar a parte inteira da parte fracionária.



O comando print também funciona com variáveis:

```
>>> print mensagem
E aí, Doutor?
>>> print n
17
>>> print pi
3.14159
```

Em cada um dos casos, o resultado é o valor da variável. Variáveis também têm tipo. Novamente, podemos perguntar ao interpretador quais são eles:

```
>>> type(mensagem)
<type 'string'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

O tipo de uma variável é o tipo do valor ao qual ela se refere.

4.3 2.3 Nomes de variáveis e palavras reservadas

Os programadores geralmente escolhem nomes significativos para suas variáveis, pois os nomes documentam para o que a variável é usada.

Nomes de variáveis podem ser arbitrariamente longos. Eles podem conter tanto letras quanto números, mas têm de começar com uma letra. Embora seja válida a utilização de letras maiúsculas, por convenção, não usamos. Se você o fizer, lembre-se de que maiúsculas e minúsculas são diferentes. `Bruno` e `bruno` são variáveis diferentes.

O caractere para sublinhado (_) pode aparecer em um nome. Ele é muito utilizado em nomes com múltiplas palavras, tal como em `meu_nome` ou `preco_do_cha_na_china`.

Se você der a uma variável um nome inválido, causará um erro de sintaxe:

```
>>> 76trombones = "grande parada"
SyntaxError: invalid syntax
>>> muito$ = 1000000
SyntaxError: invalid syntax
>>> class = "Ciencias da Computacao 101"
SyntaxError: invalid syntax
```

`76trombones` é inválida por não começar com uma letra. `muito$` é inválida por conter um caractere ilegal, o cifrão. Mas o que está errado com `class`?

Ocorre que `class` é uma das **palavras reservadas** em Python. Palavras reservadas definem as regras e a estrutura da linguagem e não podem ser usadas como nomes de variáveis.

Python tem 29 palavras reservadas:

```
and       def       exec      if        not       return
assert    del       finally   import    or        try
break     elif      for       in        pass      while
class     else      from      is        print     yield
continue  except    global    lambda    raise
```

Pode ser útil ter essa lista à mão [2]. Se o interpretador acusar erro sobre um de seus nomes de variável e você não souber o porquê, veja se o nome está na lista.

4.4 2.4 Comandos

Um comando é uma instrução que o interpretador Python pode executar. Vimos até agora dois tipos de comandos: de exibição (`print`) e de atribuição.

Quando você digita um comando na linha de comando, o Python o executa e mostra o resultado, se houver um. O resultado de um comando `print` é a exibição de um valor. Comandos de atribuição não produzem um resultado visível.

Um *script* normalmente contém uma sequência de comandos. Se houver mais de um comando, os resultados aparecerão um de cada vez, conforme cada comando seja executado.

Por exemplo, o "script":

```
print 1
x = 2
print 2
```

produz a saída:

```
1
2
```

Lembrando que o comando de atribuição não produz saída.

4.5 2.5 Avaliando expressões

Uma expressão é uma combinação de valores, variáveis e operadores. Se você digitar uma expressão na linha de comando, o interpretador **avalia** e exibe o resultado:

```
>>> 1 + 1
2
```

Embora expressões contenham valores, variáveis e operadores, nem toda expressão contém todos estes elementos. Um valor por si só é considerado uma expressão, do mesmo modo que uma variável:

```
>>> 17
17
>>> x
2
```

[2] N.T.: esta lista pode ser obtida através do próprio interpretador Python, com apenas dois comandos: `import keyword; print keyword.kwlist`

Avaliar uma expressão não é exatamente a mesma coisa que imprimir um valor:

```
>>> mensagem = "E aí, Doutor?"
>>> mensagem
'E aí, Doutor?'
>>> print mensagem
E aí, Doutor?
```

Quando Python exibe o valor de uma expressão, usa o mesmo formato que você usaria para entrar com o valor. No caso de strings, isso significa que as aspas são incluídas [3]. Mas o comando `print` imprime o valor da expressão, que, neste caso, é o conteúdo da string.

Num *script*, uma expressão sozinha é um comando válido, porém sem efeito. O *script*:

```
17
3.2
"Alô, Mundo!"
1 + 1
```

não produz qualquer saída. Como você mudaria o "script" para exibir os valores destas quatro expressões?

4.6 2.6 Operadores e operandos

Operadores são símbolos especiais que representam computações como adição e multiplicação. Os valores que o operador usa são chamados **operandos**.

Todas as expressões seguintes são válidas em Python e seus significados são mais ou menos claros:

```
20+32   hora-1   hora*60+minuto   minuto/60   5**2   (5+9)*(15-7)
```

Em Python, os símbolos +, -, / e o uso de parênteses para agrupamento têm o mesmo significado que em matemática. O asterisco (`*`) é o símbolo para multiplicação e `**` é o símbolo para potenciação.

Quando um nome de variável aparece no lugar de um operando, ele é substituído pelo valor da variável, antes da operação ser executada.

Adição, subtração, multiplicação e potenciação fazem o que se espera, mas você pode ficar surpreso com a divisão. A operação seguinte tem um resultado inesperado:

```
>>> minuto = 59
>>> minuto/60
0
```

O valor de minuto é 59 e, em aritmética convencional, 59 dividido por 60 é 0,98333, não 0. A razão para a discrepância é que Python está realizando uma **divisão inteira**.

Quando ambos os operandos são inteiros, o resultado tem de ser também um inteiro e, por convenção, a divisão inteira sempre arredonda para baixo, mesmo em casos como este, em que o inteiro seguinte está muito próximo:

```
>>> minuto*100/60
98
```

De novo, o resultado é arredondado para baixo, mas agora pelo menos a resposta é aproximadamente correta. A alternativa é usar a divisão em ponto flutuante, o que veremos no capítulo 3.

[3] N.T.: Python aceita aspas simples ou duplas para delimitar strings.

4.7 2.7 Ordem dos operadores

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das **regras de precedência**. Python segue as mesmas regras de precedência para seus operadores matemáticos que a matemática. O acrônimo **PEMDAS** é uma maneira prática de lembrar a ordem das operações:

- **P**: Parênteses têm a mais alta precedência e podem ser usados para forçar uma expressão a ser avaliada na ordem que você quiser. Já que expressões entre parênteses são avaliadas primeiro, `2 * (3-1)` é 4, e `(1+1)**(5-2)` é 8. Você também pode usar parênteses para tornar uma expressão mais fácil de ler, como em `(minuto * 100) / 60`, ainda que isso não altere o resultado.
- **E**: Exponenciação ou potenciação tem a próxima precedência mais alta, assim `2**1+1` é 3 e não 4, e `3*1**3` é 3 e não 27.
- **MDAS**: Multiplicação e Divisão têm a mesma precedência, que é mais alta do que a da Adição e da Subtração, que também têm a mesma precedência. Assim `2*3-1` dá 5 em vez de 4, e `2/3-1` é `-1`, não 1 (lembre-se de que na divisão inteira, `2/3=0`).
- Operadores com a mesma precedência são avaliados da esquerda para a direita. Assim, na expressão `minuto*100/60`, a multiplicação acontece primeiro, resultando em `5900/60`, o que se transforma produzindo `98`. Se as operações tivessem sido avaliadas da direita para a esquerda, o resultado poderia ter sido `59*1`, que é `59`, que está errado.

4.8 2.8 Operações com strings

De maneira geral, você não pode executar operações matemáticas em strings, ainda que as strings se pareçam com números. O que segue é inválido (assumindo que `mensagem` é do tipo `string`):

```
mensagem-1
"Alô"/123
mensagem*"Alô"
"15"+2
```

Interessante é o operador `+`, que funciona com strings, embora ele não faça exatamente o que você poderia esperar. Para strings, o operador + representa **concatenação**, que significa juntar os dois operandos ligando-os pelos extremos. Por exemplo:

```
fruta = "banana"
assada = " com canela"
print fruta + assada
```

A saída deste programa é `banana com canela`. O espaço antes da palavra `com` é parte da string e é necessário para produzir o espaço entre as strings concatenadas.

O operador `*` também funciona com strings; ele realiza repetição. Por exemplo, `"Legal"*3` é `"LegalLegaLegal"`. Um dos operadores tem que ser uma string; o outro tem que ser um inteiro.

Por um lado, esta interpretação de + e `*` faz sentido pela analogia entre adição e multiplicação. Assim como `4*3` equivale a `4+4+4`, não é de estranhar que `"Legal"*3` seja o mesmo que `"Legal"+"Legal"+"Legal"`. Por outro lado, uma diferença significativa separa concatenação e repetição de adição e multiplicação. Você saberia mencionar uma propriedade da adição e da multiplicação que não ocorre na concatenação e na repetição?

4.9 2.9 Composição

Até agora, vimos os elementos de um programa (variáveis, expressões, e instruções ou comandos) isoladamente, sem mencionar como combiná-los.

Uma das características mais práticas das linguagens de programação é a possibilidade de pegar pequenos blocos e combiná-los numa **composição**. Por exemplo, nós sabemos como somar números e sabemos como exibi-los; acontece que podemos fazer as duas coisas ao mesmo tempo:

```
>>> print 17 + 3
20
```

Na realidade, a soma tem que acontecer antes da impressão, assim, as ações não estão na realidade acontecendo ao mesmo tempo. O ponto é que qualquer expressão envolvendo números, strings, e variáveis pode ser usada dentro de um comando `print`. Você já tinha visto um exemplo disto:

```
print "Número de minutos desde a meia-noite: ", hora*60+minuto
```

Esta possibilidade pode não parecer muito impressionante agora, mas você verá outros exemplos em que a composição torna possível expressar cálculos e tarefas complexas de modo limpo e conciso.

Atenção: Existem limites quanto ao lugar onde você pode usar certos tipos de expressão. Por exemplo, o lado esquerdo de um comando de atribuição tem que ser um *nome de variável*, e não uma expressão. Assim, o seguinte não é válido: `minuto+1 = hora`.

4.10 2.11 Glossário

atribuição (*assignment*) Comando que atribui um valor a uma variável.

avaliar (*evaluate*) Simplificar uma expressão através da realização de operações, para produzir um valor único.

comando (*statement*) Trecho de código que representa uma instrução ou ação. Até agora, os comandos vistos foram de atribuição e exibição.

comentário (*comment*) Informação em um programa dirigida a outros programadores (ou qualquer pessoa que esteja lendo o código fonte) e que não tem efeito na execução do programa.

composição (*composition*) Habilidade de combinar expressões e comandos simples em expressões e comandos compostos, de forma a representar operações complexas de forma concisa.

concatenar (*concatenate*) Juntar dois operandos lado a lado.

diagrama de estado (*state diagram*) Representação gráfica de um conjunto de variáveis e os valores aos quais elas se referem.

divisão inteira (*integer division*) Operação que divide um inteiro por outro e resulta em um inteiro. A divisão inteira resulta no número de vezes que o numerador é divisível pelo denominador e descarta qualquer resto.

expressão (*expression*) Combinação de variáveis, operadores e valores, que representa um resultado único.

operando (*operand*) Um dos valores sobre o qual o operador opera.

operador (*operator*) Símbolo especial que representa uma computação simples, como adição, multiplicação ou concatenação de strings.

palavra-chave (*keyword*) Palavra reservada usada pelo compilador/interpretador para analisar o programa; você não pode usar palavras-chave como `if`, `def`, e `while` como nomes de variáveis.

ponto-flutuante (*floating-point*) Formato para representar números que possuem partes fracionárias.

regras de precedência (*rules of precedence*) O conjunto de regras que governa a ordem em que expressões envolvendo múltiplos operadores e operandos são avaliadas.

tipo (*type*) Um conjunto de valores. O tipo de um valor determina como ele pode ser usado em expressões. Até agora, os tipos vistos são: inteiros (tipo `int`), números em ponto-flutuante (tipo `float`) e strings (tipo `string`).

valor (*value*) Um número ou string (ou outra coisa que ainda vamos conhecer) que pode ser atribuída a uma variável ou computada em uma expressão.

variável (*variable*) Nome que se refere a um valor.

CAPÍTULO 5

Capítulo 3: Funções

Tópicos

5.1 3.1 Chamadas de funções

Você já viu um exemplo de uma **chamada de função**:

```
>>> type('32')
<type 'str'>
```

O nome da função é `type` e ela exibe o tipo de um valor ou variável. O valor ou variável, que é chamado de **argumento** da função, tem que vir entre parênteses. É comum se dizer que uma função 'recebe' um valor ou mais valores e 'retorna' um resultado. O resultado é chamado de **valor de retorno**.

Em vez de imprimir um valor de retorno, podemos atribui-lo a uma variável:

```
>>> bia = type('32')
>>> print bia
<type 'str'>
```

Como outro exemplo, a função `id` recebe um valor ou uma variável e retorna um inteiro, que atua como um identificador único para aquele valor:

```
>>> id(3)
134882108
>>> bia = 3
>>> bia(beth)
134882108
```

Todo valor tem um `id`, que é um número único relacionado ao local onde ele está guardado na memória do computador. O `id` de uma variável é o `id` do valor a qual ela se refere.

5.2 3.2 Conversão entre tipos

Python provê uma coleção de funções nativas que convertem valores de um tipo em outro. A função `int` recebe um valor e o converte para inteiro, se possível, ou, se não, reclama:

```
>>> int('32')
32
>>> int('Alô')
ValueError: invalid literal for int() : Alô
```

`int` também pode converter valores em ponto flutuante para inteiro, mas lembre que isso trunca a parte fracionária:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

A função `float` converte inteiros e strings em números em ponto flutuante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, a função `str` converte para o tipo `string`:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Pode parecer curioso que Python faça distinção entre o valor inteiro `1` e o valor em ponto flutuante `1.0`. Eles podem representar o mesmo número, mas pertencem a tipos diferentes. A razão é que eles são representados de modo diferente dentro do computador.

5.3 3.3 Coerção entre tipos

Agora que podemos converter entre tipos, temos outra maneira de lidar com a divisão inteira. Voltando ao exemplo do capítulo anterior, suponha que queiramos calcular a fração de hora que já passou. A expressão mais óbvia, `minuto / 60`, faz aritmética inteira, assim, o resultado é sempre 0, mesmo aos 59 minutos passados da hora.

Uma solução é converter `minuto` para ponto flutuante e fazer a divisão em ponto flutuante:

```
>>> minuto = 59
>>> float(minuto) / 60
0.983333333333
```

Opcionalmente, podemos tirar vantagem das regras de conversão automática entre tipos, chamada de **coerção de tipos**. Para os operadores matemáticos, se qualquer operando for um `float`, o outro é automaticamente convertido para `float`:

```
>>> minuto = 59
>>> minuto / 60.0
0.983333333333
```

Fazendo o denominador um `float`, forçamos o Python a fazer a divisão em ponto flutuante.

5.4 3.4 Funções matemáticas

Em matemática, você provavelmente já viu funções como seno (`sen`, `sin` em inglês) e logaritmo (`log`), e aprendeu a resolver expressões como `sen(pi/2)` e `log(1/x)`. Primeiro você resolve e expressão entre parênteses (o argumento). Por exemplo, `pi/2` é aproximadamente 1,571, e `1/x` é 0.1 (se `x` for 10,0).

Aí você avalia a função propriamente dita, seja procurando numa tabela ou realizando vários cálculos. O `sen` de 1,571 é 1 e o `log` de 0,1 é -1 (assumindo que `log` indica o logaritmo na base 10).

Este processo pode ser aplicado repetidamente para avaliar expressões mais complicadas, como `log(1/sen(pi/2))`. Primeiro você avalia o argumento na função mais interna, depois avalia a função e assim por diante.

Python tem um módulo matemático que provê a maioria das funções matemáticas mais familiares. Um **módulo** é um arquivo que contém uma coleção de funções relacionadas agrupadas juntas.

Antes de podermos usar as funções contidas em um módulo, temos de importá-lo:

```
>>> import math
```

Para chamar uma das funções, temos que especificar o nome do módulo e o nome da função, separados por um ponto. Esse formato é chamado de **notação de ponto**:

```
>>> decibel = math.log10(17.0)
>>> angulo = 1.5
>>> altura = math.sin(angulo)
```

A primeira instrução atribui a `decibel` o logaritmo de 17 na base 10. Existe também uma função chamada `log`, usada para calcular o logaritmo em outra base ou o logaritmo natural de um número (base `e`).

A terceira instrução encontra o seno do valor da variável `angulo`. `sin` e as outras funções trigonométricas (`cos`, `tan`, etc.) recebem argumentos em radianos. Para converter de graus em radianos, divida por 360 e multiplique por `2*pi`. Por exemplo, para encontrar o seno de 45 graus, primeiro calcule o ângulo em radianos e depois ache o seno:

```
>>> graus = 45
>>> angulo = graus * 2 * math.pi / 360.0
>>> math.sin(angulo)
0.707106781187
```

A constante pi também é parte do módulo `math`. Se você sabe geometria, pode checar o resultado anterior comparando-o com a raiz quadrada de dois dividido por dois:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

5.5 3.5 Composição

Do mesmo modo como nas funções matemáticas, as funções do Python podem ser compostas, o que significa que você pode usar uma expressão como parte de outra. Por exemplo, você pode usar qualquer expressão como um argumento para uma função:

```
>>> x = math.cos(angulo + pi/2)
```

Esta instrução toma o valor de `pi`, divide-o por 2, e soma o resultado ao valor de `angulo`. A soma é então passada como um argumento para a função `cos`.

Você também pode pegar o resultado de uma função e passá-lo como um argumento para outra:

```
>>> x = math.exp(math.log(10.0))
```

Esta instrução encontra o logaritmo natural (base `e`) de 10 e então eleva `e` àquela potência. O resultado é atribuído a `x`.

5.6 3.6 Adicionando novas funções

Até aqui, temos utilizado somente as funções que vêm com Python, mas também é possível adicionar novas funções. Criar novas funções para resolver seus próprios problemas é uma das coisas mais úteis de uma linguagem de programação de propósito geral.

No contexto de programação, **função** é uma sequência nomeada de instruções ou comandos, que realizam uma operação desejada. Esta operação é especificada numa **definição de função**. Até agora, as funções que usamos neste livro são pré-definidas e suas definições não foram apresentadas. Isso demonstra que podemos usar funções sem ter que nos preocupar com os detalhes de suas definições.

A sintaxe para uma definição de função é:

```
def NOME_DA_FUNCAO( LISTA DE PARAMETROS ) :
  COMANDOS
```

Você pode usar o nome que quiser para as funções que criar, exceto as palavras reservadas do Python. A lista de parâmetros especifica que informação, se houver alguma, você tem que fornecer para poder usar a nova função.

Uma função pode ter quantos comandos forem necessários, mas eles precisam ser endentados a partir da margem esquerda. Nos exemplos deste livro, usaremos uma endentação de dois espaços.

As primeiras funções que vamos mostrar não terão parâmetros, então, a sintaxe terá esta aparência:

```
def novaLinha():
  print
```

Esta função é chamada de `novaLinha`. Os parênteses vazios indicam que ela não tem parâmetros. Contém apenas um único comando, que gera como saída um caractere de nova linha (isso é o que acontece quando você usa um comando `print` sem qualquer argumento).

A sintaxe para a chamada desta nova função é a mesma sintaxe para as funções nativas:

```
print 'Primeira Linha.'
novaLinha()
print 'Segunda Linha.'
```

A saída deste programa é:

```
Primeira Linha.

Segunda Linha.
```

Observe o espaço extra entre as duas linhas. E se quiséssemos mais espaço entre as linhas? Poderíamos chamar a mesma função repetidamente:

```
print 'Primeira Linha.'
novaLinha()
novaLinha()
novaLinha()
print 'Segunda Linha.'
```

Ou poderíamos escrever uma nova função chamada `tresLinhas`, que produzisse três novas linhas:

```
def tresLinhas() :
  novaLinha()
  novaLinha()
  novaLinha()

print 'Primeira Linha.'
tresLinhas()
print 'Segunda Linha.'
```

Esta função contém três comandos, todos com recuo de dois espaços a partir da margem esquerda. Já que o próximo comando não está endentado, Python reconhece que ele não faz parte da função.

Algumas coisas que devem ser observadas sobre este programa:

1. Você pode chamar o mesmo procedimento repetidamente. Isso é muito comum, além de útil.
2. Você pode ter uma função chamando outra função; neste caso `tresLinhas` chama `novaLinha`.

Pode não estar claro, até agora, de que vale o esforço de criar novas funções - existem várias razões, mas este exemplo demonstra duas delas:

- Criar uma nova função permite que você coloque nome em um grupo de comandos. As funções podem simplificar um programa ao ocultar a execução de uma tarefa complexa por trás de um simples comando cujo nome pode ser uma palavra em português, em vez de algum código misterioso.
- Criar uma nova função pode tornar o programa menor, por eliminar código repetido. Por exemplo, um atalho para 'imprimir' nove novas linhas consecutivas é chamar `tresLinhas` três vezes.

> Como exercício, escreva uma função chamada `noveLinhas` que use `tresLinhas` para imprimir nove linhas em branco. Como você poderia imprimir vinte e sete novas linhas?

5.7 3.7 Definições e uso

Reunindo os fragmentos de código da Seção 3.6, o programa completo fica assim:

```
def novaLinha() :
  print

def tresLinhas() :
  novaLinha()
  novaLinha()
  novaLinha()

print 'Primeira Linha.'
```

```
tresLinhas()
print 'Segunda Linha.'
```

Esse programa contém duas definições de funções: `novaLinha` e `tresLinhas`. Definições de funções são executadas como quaisquer outros comandos, mas o efeito é criar a nova função. Os comandos dentro da definição da função não são executados até que a função seja chamada, logo, a definição da função não gera nenhuma saída.

Como você já deve ter imaginado, é preciso criar uma função antes de poder executá-la. Em outras palavras, a definição da função tem que ser executada antes que ela seja chamada pela primeira vez.

> Como exercício, mova as últimas três linhas deste programa para o topo, de modo que a chamada da função apareça antes das definições. Rode o programa e veja que mensagem de erro você terá.
>
> Também a título de exercício, comece com a versão que funciona do programa e mova a definição de `novaLinha` para depois da definição de `tresLinhas`. O que acontece quando você roda este programa?

5.8 3.8 Fluxo de execução

Para assegurar que uma função esteja definida antes do seu primeiro uso, é preciso saber em que ordem os comandos são executados, ou seja, descobrir qual o **fluxo de execução** do programa.

A execução sempre começa com o primeiro comando do programa. Os comandos são executados um de cada vez, pela ordem, de cima para baixo.

As definições de função não alteram o fluxo de execução do programa, mas lembre-se que comandos dentro da função não são executados até a função ser chamada. Embora não seja comum, você pode definir uma função dentro de outra. Neste caso, a definição mais interna não é executada até que a função mais externa seja chamada.

Chamadas de função são como um desvio no fluxo de execução. Em vez de ir para o próximo comando, o fluxo salta para a primeira linha da função chamada, executa todos os comandos lá e então volta atrás para retomar de onde havia deixado.

Parece muito simples, até a hora em que você lembra que uma função pode chamar outra. Enquanto estiver no meio de uma função, o programa poderia ter de executar os comandos em uma outra função. Mas enquanto estivesse executando esta nova função, o programa poderia ter de executar ainda outra função!

Felizmente, Python é adepto de monitorar a posição onde está, assim, cada vez que uma função se completa, o programa retoma de onde tinha parado na função que a chamou. Quando chega ao fim do programa, ele termina.

Qual a moral dessa história sórdida? Quando você for ler um programa, não o leia de cima para baixo. Em vez disso, siga o fluxo de execução.

5.9 3.9 Parâmetros e argumentos

Algumas das funções nativas que você já usou requerem argumentos, aqueles valores que controlam como a função faz seu trabalho. Por exemplo, se você quer achar o seno de um número, você tem que indicar qual número é. Deste modo, `sin` recebe um valor numérico como um argumento.

Algumas funções recebem mais de um argumento. Por exemplo, `pow` recebe dois argumentos, a base e o expoente. Dentro da função, os valores que lhe são passados são atribuídos a variáveis chamadas **parâmetros**.

Veja um exemplo de uma função definida pelo usuário, que recebe um parâmetro:

```
def imprimeDobrado(bruno):
  print bruno, bruno
```

Esta função recebe um único argumento e o atribui a um parâmetro chamado `bruno`. O valor do parâmetro (a essa altura, não sabemos qual será) é impresso duas vezes, seguido de uma nova linha. Estamos usando `bruno` para mostrar que o nome do parâmetro é decisão sua, mas claro que é melhor escolher um nome que seja mais ilustrativo.

A função `imprimeDobrado` funciona para qualquer tipo que possa ser impresso:

```
>>> imprimeDobrado('Spam')
Spam Spam
>>> imprimeDobrado(5)
5 5
>>> imprimeDobrado(3.14159)
3.14159 3.14159
```

Na primeira chamada da função, o argumento é uma string. Na segunda, é um inteiro. Na terceira é um `float`.

As mesmas regras de composição que se aplicam a funções nativas também se aplicam às funções definidas pelo usuário, assim, podemos usar qualquer tipo de expressão como um argumento para `imprimeDobrado`:

```
>>> imprimeDobrado('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> imprimeDobrado(math.cos(math.pi))
-1.0 -1.0
```

Como acontece normalmente, a expressão é avaliada antes da execução da função, assim `imprimeDobrado` imprime `SpamSpamSpamSpam SpamSpamSpamSpam` em vez de `'Spam'*4 'Spam'*4`.

> Como exercício, escreva um chamada a `imprimeDobrado` que imprima `'Spam'*4 'Spam'*4`. Dica: strings podem ser colocadas tanto entre aspas simples quanto duplas e o tipo de aspas que não for usado para envolver a string pode ser usado dentro da string, como parte dela.

Também podemos usar uma variável como argumento:

```
>>> miguel = 'Eric, the half a bee.'
>>> imprimeDobrado(miguel)
Eric, the half a bee. Eric, the half a bee.
```

N.T.: "Eric, the half a bee" é uma música do grupo humorístico britânico Monty Python. A linguagem Python foi batizada em homenagem ao grupo e, por isso, os programadores gostam de citar piadas deles em seus exemplos.

Repare numa coisa importante: o nome da variável que passamos como um argumento (`miguel`) não tem nada a ver com o nome do parâmetro (`bruno`). Não importa de que modo o valor foi chamado de onde veio (do 'chamador'); aqui, em `imprimeDobrado`, chamamos a todo mundo de `bruno`.

5.10 3.10 Variáveis e parâmetros são locais

Quando você cria uma **variável local** dentro de uma função, ela só existe dentro da função e você não pode usá-la fora de lá. Por exemplo:

```
def concatDupla(parte1, parte2):
  concat = parte1 + parte2
  imprimeDobrado(concat)
```

Esta função recebe dois argumentos, concatena-os, e então imprime o resultado duas vezes. Podemos chamar a função com duas strings:

```
>>> canto1 = 'Pie Jesu domine, '
>>> canto2 = 'dona eis requiem. '
>>> concatDupla(canto1, canto2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.
```

Quando a função concatDupla termina, a variável concat é destruída. Se tentarmos imprimi-la, teremos um erro:

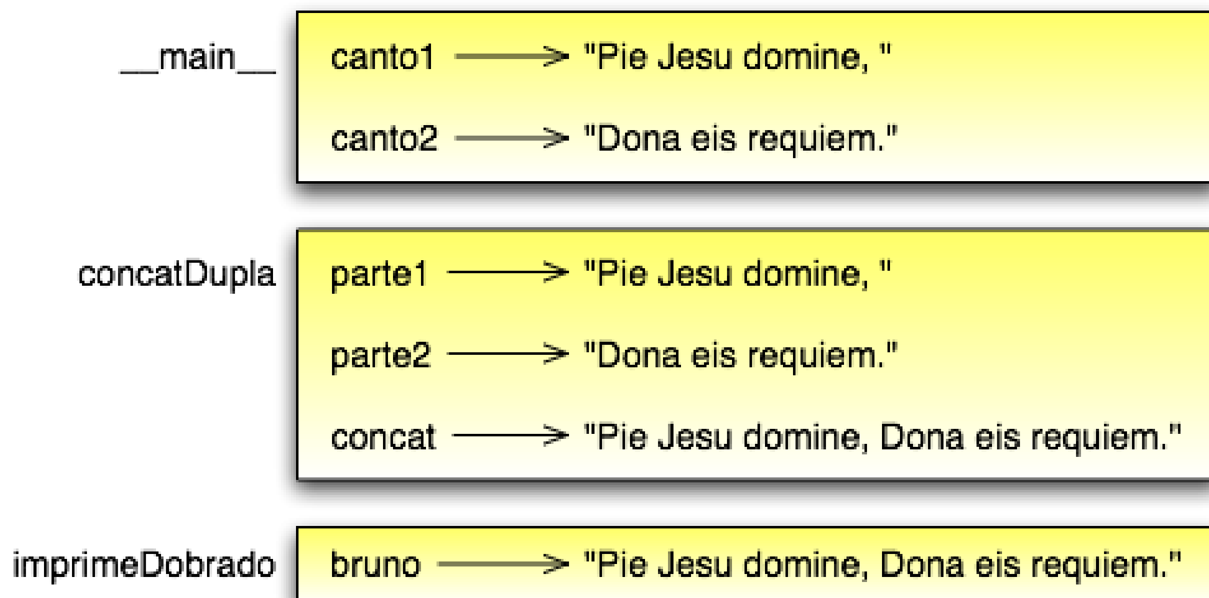
```
>>> print concat
NameError: concat
```

Parâmetros são sempre locais. Por exemplo, fora da função imprimeDobrado, não existe nada que se chama bruno. Se você tentar utilizá-la, o Python vai reclamar.

5.11 3.11 Diagramas da pilha

Para entender que variáveis podem ser usadas aonde, às vezes é útil desenhar um **diagrama da pilha**. Como os diagramas de estado, diagramas da pilha mostram o valor de cada variável, mas também a função à qual cada variável pertence.

Cada função é representada por um **quadro**. Um quadro é uma caixa com o nome de uma função ao lado dela e os parâmetros e variáveis da função dentro dela. O diagrama de pilha para o exemplo anterior tem a seguinte aparência:



A ordem da pilha mostra o fluxo de execução. imprimeDobrado foi chamado por concatDupla, e concatDupla foi chamado por __main__ (principal), que é um nome especial para a função mais no topo. Quando você cria uma variável fora de qualquer função, ela pertence à __main__.

Cada parâmetro se refere ao mesmo valor que o seu argumento correspondente. Assim, parte1 tem o mesmo valor de canto1, parte2 tem o mesmo valor de canto2 e bruno tem o mesmo valor de concat.

Se um erro acontece durante uma chamada de função, Python imprime o nome da função, e o nome da função que a chamou, e o nome da função que chamou a que chamou, percorrendo todo o caminho de volta a __main__.

Por exemplo, se tentássemos acessar concat de dentro de imprimeDobrado, teríamos um NameError:

```
Traceback (innermost last):
  File "teste.py", line 13, in __main__
    concatDupla(canto1, canto2)
  File "teste.py", line 5, in concatDupla
    imprimeDobrado(concat)
  File "teste.py", line 9, in imprimeDobrado
```

```
    print concat
NameError: concat
```

Esta lista de funções é chamada de **traceback**. Ela mostra em qual arquivo de programa o erro ocorreu, em que linha, e quais funções estavam sendo executadas naquele momento. Mostra também a linha de código que causou o erro.

Note a similaridade entre o traceback e o diagrama da pilha. Não é coincidência.

5.12 3.12 Funções com resultados

A essa altura, você deve ter percebido que algumas das funções que estamos usando, tais como as funções matemáticas, produzem resultados. Outras funções, como `novaLinha`, executam uma ação, mas não retornam um valor. O que levanta algumas questões:

1. O que acontece se você chama uma função e não faz nada com o resultado (por exemplo, não atribui o resultado a uma variável ou o usa como parte de uma expressão maior)?
2. O que acontece se você usa uma função que não produz resultado em uma expressão tal como `novaLinha() + 7`?
3. Você pode escrever funções que produzem resultados, ou está preso a funções como `novaLinha` e `imprimeDobrado`?

A resposta para a terceira questão é afirmativa e nós vamos fazer isso no Capítulo 5.

> A título de exercício, responda as outras duas questões testando-as. Se tiver dúvida sobre o que é válido ou inválido em Python, tente buscar a resposta perguntando ao interpretador.

5.13 3.13 Glossário

argumento (*argument*) Valor fornecido a uma função quando ela é chamada. Este valor é atribuído ao parâmetro correspondente na função.

chamada de função (*function call*) Comando que executa uma função. Consiste do nome da função seguido de uma lista de argumentos entre parênteses.

coerção de tipo (*type coercion*) Uma coerção de tipo que ocorre automaticamente, de acordo com as regras de coercividade do Python.

conversão de tipo (*type conversion*) Comando explícito que pega um valor de um tipo e devolve o valor correspondente em outro tipo.

definição de função (*function definition*) Comando que cria uma nova função, especificando seu nome, parâmetros e comandos que ela executa.

diagrama da pilha (*stack diagram*) Representação gráfica da pilha de funções, suas variáveis e os valores aos quais elas se referem.

fluxo de execução (*flow of execution*) A ordem na qual os comandos são executados durante a execução do programa.

frame Retângulo no diagrama da pilha que representa uma chamada de função. Contém as variáveis locais e os parâmetros da função.

função (*function*) Sequência de comandos nomeada, que realiza alguma tarefa útil. As funções podem ou não receber parâmetros e podem ou não retornar valores.

módulo (*module*) Arquivo que contém uma coleção de funções e classes relacionadas entre si.

notação de ponto (*dot notation*) A sintaxe para chamar uma função que está em outro módulo, especificando o nome do módulo, seguido por um ponto (.) e o nome da função.

parâmetro (*parameter*) Nome usado numa função para referir-se a um valor passado como argumento.

traceback Lista de funções que estão em execução, impressa quando um erro de execução ocorre.

valor de retorno (*return value*) O resultado da função. Se uma chamada de função é usada como expressão, o valor de retorno é o valor da expressão.

variável local (*local variable*) Variável definida dentro da função. Uma variável local só pode ser usada dentro da função onde foi definida.

Capítulo 4: Condicionais e recursividade

Tópicos

6.1 4.1 O operador módulo

O **operador módulo** trabalha com inteiros (e expressões que têm inteiros como resultado) e produz o resto da divisão do primeiro pelo segundo. Em Python, o operador módulo é um símbolo de porcentagem (`%`). A sintaxe é a mesma que a de outros operadores:

```
>>> quociente = 7 / 3
>>> print quociente
2
>>> resto = 7 % 3
>>> print resto
1
```

Então, 7 dividido por 3 é 2 e o resto é 1.

O operador módulo se revela surpreendentemente útil. Por exemplo, você pode checar se um número é divisível por outro - se `x % y` dá zero, então `x` é divisível por `y`.

Você também pode extrair o algarismo ou algarismos mais à direita de um número. Por exemplo, `x % 10` resulta o algarismo mais à direita de `x` (na base 10). Similarmente, `x % 100` resulta nos dois dígitos mais à direita.

6.2 4.2 Expressões booleanas

Uma expressão booleana é uma expressão que é verdadeira (*True*) ou é falsa (*False*). Em Python, uma expressão que é verdadeira tem o valor 1, e uma expressão que é falsa tem o valor 0.

O operador == compara dois valores e produz uma expressão booleana:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

No primeiro comando, os dois operadores são iguais, então a expressão avalia como `True` (verdadeiro); no segundo comando, 5 não é igual a 6, então temos `False` (falso).

O operador == é um dos **operadores de comparação**; os outros são:

```
x != y          # x é diferente de y
x > y           # x é maior que y
x < y           # x é menor que y
x >= y          # x é maior ou igual a y
x <= y          # x é menor ou igual a y
```

Embora esses operadores provavelmente sejam familiares a você, os símbolos em Python são diferentes dos símbolos da matemática. Um erro comum é usar um sinal de igual sozinho (=) em vez de um duplo (==). Lembre-se de que = é um operador de atribuição e == é um operador de comparação. Também não existem coisas como =< ou =>.

6.3 4.3 Operadores lógicos

Existem três operadores lógicos: `and`, `or`, `not` (e, ou, não). A semântica (significado) destes operadores é similar aos seus significados em inglês (ou português). Por exemplo, `x > 0 and x < 10` é verdadeiro somente se x for maior que 0 *e* menor que 10.

`n%2 == 0 or n%3 == 0` é verdadeiro se qualquer das condições for verdadeira, quer dizer, se o número `n` for divisível por 2 *ou* por 3.

Finalmente, o operador lógico `not` nega uma expressão booleana, assim, `not(x > y)` é verdadeiro se `(x > y)` for falso, quer dizer, se `x` for menor ou igual a `y`.

A rigor, os operandos de operadores lógicos deveriam ser expressões booleanas, mas Python não é muito rigoroso. Qualquer número diferente de zero é interpretado como verdadeiro (*True*):

```
>>> x = 5
>>> x and 1
1
>>> y = 0
>>> y and 1
0
```

Em geral, esse tipo de coisa não é considerado de bom estilo. Se você precisa comparar um valor com zero, deve fazê-lo explicitamente.

6.4 4.4 Execução condicional

Para poder escrever programas úteis, quase sempre precisamos da habilidade de checar condições e mudar o comportamento do programa de acordo com elas. As **instruções condicionais** nos dão essa habilidade. A forma mais simples é a instrução `if` (se):

```
if x > 0
  print "x é positivo"
```

A expressão booleana depois da instrução `if` é chamada de **condição**. Se ela é verdadeira (*true*), então a instrução endentada é executada. Se não, nada acontece.

Assim como outras instruções compostas, a instrução `if` é constituída de um cabeçalho e de um bloco de instruções:

```
CABECALHO:
  PRIMEIRO COMANDO
  ...
  ULTIMO COMANDO
```

O cabeçalho começa com uma nova linha e termina com dois pontos (`:`). Os comandos ou instruções endentados que seguem são chamados de **bloco**. A primeira instrução não endentada marca o fim do bloco. Um bloco de comandos dentro de um comando composto ou instrução composta é chamado de **corpo** do comando.

Não existe limite para o número de instruções que podem aparecer no corpo de uma instrução `if`, mas tem que haver pelo menos uma. Ocasionalmente, é útil ter um corpo sem nenhuma instrução (usualmente, como um delimitador de espaço para código que você ainda não escreveu). Nesse caso, você pode usar o comando `pass`, que indica ao Python: "passe por aqui sem fazer nada".

6.5 4.5 Execução alternativa

Um segundo formato da instrução `if` é a execução alternativa, na qual existem duas possibilidades e a condição determina qual delas será executada. A sintaxe se parece com:

```
if x % 2 == 0:
  print x, "é par"
else:
  print x, "é impar"
```

Se o resto da divisão de x por 2 for 0, então sabemos que x é par, e o programa exibe a mensagem para esta condição. Se a condição é falsa, o segundo grupo de instruções é executado. Desde que a condição deva ser verdadeira (*True*) ou falsa (*False*), precisamente uma das alternativas vai ser executada. As alternativas são chamadas **ramos** (*branches*), porque existem ramificações no fluxo de execução.

Por final, se você precisa checar a paridade de números com frequência, pode colocar este código dentro de uma função:

```
def imprimeParidade(x):
  if x % 2 == 0:
    print x, "é par"
  else:
    print x, "é impar"
```

Para qualquer valor de `x`, `imprimeParidade` exibe uma mensagem apropriada. Quando você a chama, pode fornecer uma expressão de resultado inteiro como um argumento:

```
>>> imprimeParidade(17)
>>> imprimeParidade(y+1)
```

6.6 4.6 Condicionais encadeados

Às vezes existem mais de duas possibilidades e precisamos de mais que dois ramos. Uma **condicional encadeada** é uma maneira de expressar uma operação dessas:

```
if x < y:
  print x, "é menor que", y
elif x > y:
  print x, "é maior que", y
else:
  print x, "e", y, "são iguais"
```

`elif` é uma abreviação de "else if" ("senão se"). De novo, precisamente um ramo será executado. Não existe limite para o número de instruções `elif`, mas se existir uma instrução `else` ela tem que vir por último:

```
if escolha == 'A':
  funcaoA()
elif escolha == 'B':
  funcaoB()
elif escolha == 'C':
  funcaoC()
else:
  print "Escolha inválida."
```

Cada condição é checada na ordem. Se a primeira é falsa, a próxima é checada, e assim por diante. Se uma delas é verdadeira, o ramo correspondente é executado, e a instrução termina. Mesmo que mais de uma condição seja verdadeira, apenas o primeiro ramo verdadeiro executa.

> Como exercício, coloque os exemplos acima em funções chamadas `comparar(x, y)` e `executar(escolha)`.

6.7 4.7 Condicionais aninhados

Um condicional também pode ser aninhado dentro de outra. Poderíamos ter escrito o exemplo tricotômico (dividido em três) como segue:

```
if x == y:
  print x, "e", y, "são iguais"
else:
  if x < y:
    print x, "é menor que", y
  else:
    print x, "é maior que", y
```

O condicional mais externo tem dois ramos. O primeiro ramo contém uma única instrução de saída. O segundo ramo contém outra instrução if, que por sua vez tem dois ramos. Os dois ramos são ambos instruções de saída, embora pudessem conter instruções condicionais também.

Embora a endentação das instruções torne a estrutura aparente, condicionais aninhados tornam-se difíceis de ler rapidamente. Em geral, é uma boa ideia evitar o aninhamento quando for possível.

Operadores lógicos frequentemente fornecem uma maneira de simplificar instruções condicionais aninhadas. Por exemplo, podemos reescrever o código a seguir usando uma única condicional:

```
if 0 < x:
  if x < 10:
    print "x é um número positivo de um só algarismo."
```

A instrução `print` é executada somente se a fizermos passar por ambos os condicionais, então, podemos usar um operador `and`:

```
if 0 < x and x < 10:
  print "x é um número positivo de um só algarismo."
```

Esses tipos de condições são comuns, assim, Python provê uma sintaxe alternativa que é similar à notação matemática:

```
if 0 < x < 10:
  print "x é um número positivo de um só algarismo."
```

6.8 4.8 A instrução `return`

O comando return permite terminar a execução de uma função antes que ela alcance seu fim. Uma razão para usá-lo é se você detectar uma condição de erro:

```
import math

def imprimeLogaritmo(x):
  if x <= 0:
    print "Somente números positivos, por favor."
    return

  resultado = math.log(x)
  print "O log de x é ", resultado
```

A função `imprimeLogaritmo` recebe um parâmetro de nome `x`. A primeira coisa que ela faz é checar se `x` é menor ou igual a 0, neste caso ela exibe uma mensagem de erro e então usa `return` para sair da função. O fluxo de execução imediatamente retorna ao ponto chamador, quer dizer, de onde a função foi chamada, e as linhas restantes da função não são executadas.

Lembre-se que para usar uma função do módulo de matemática, `math`, você tem de importá-lo.

6.9 4.9 Recursividade

Já mencionamos que é válido uma função chamar outra função, e você viu vários exemplos disso. Mas ainda não tínhamos dito que também é válido uma função chamar a si mesma. Talvez não seja óbvio porque isso é bom, mas trata-se de uma das coisas mais mágicas e interessantes que um programa pode fazer. Por exemplo, dê uma olhada na seguinte função:

```
def contagemRegressiva(n):
  if n == 0:
    print "Fogo!"
  else:
    print n
    contagemRegressiva(n-1)
```

`contagemRegressiva` espera que o parâmetro, `n`, seja um inteiro positivo. Se `n` for 0, ela produz como saída a palavra "Fogo!". De outro modo, ela produz como saída `n` e então chama uma função de nome `contagemRegressiva` – ela mesma – passando `n-1` como argumento.

O que acontece se chamarmos essa função da seguinte maneira:

```
>>> contagemRegressiva(3)
```

A execução de `contagemRegressiva` começa com `n=3`, e desde que `n` não é 0, produz como saída o valor 3, e então chama a si mesma...

A execução de `contagemRegressiva` começa com `n=2`, e desde que `n` não é 0, produz como saída o valor 2, e então chama a si mesma...

A execução de `contagemRegressiva` começa com `n=1`, e desde que `n` não é 0, produz como saída o valor 1, e então chama a si mesma...

A execução de `contagemRegressiva` começa com `n=0`, e desde que `n` é 0, produz como saída a palavra "Fogo!" e então retorna.

A `contagemRegressiva` que tem `n=1` retorna.

A `contagemRegressiva` que tem `n=2` retorna.

A `contagemRegressiva` que tem `n=3` retorna.

E então estamos de volta em `__main__` (que viagem!). Assim, a saída completa se parece com:

```
3
2
1
Fogo!
```

Como um segundo exemplo, dê uma olhada novamente nas funções `novaLinha` e `tresLinhas`:

```
def novaLinha():
  print

def tresLinhas():
  novaLinha()
  novaLinha()
  novaLinha()
```

Muito embora isso funcione, não seria muito útil se precisássemos gerar como saída 2 novas linhas, ou 106. Uma alternativa melhor seria esta:

```
def nLinhas(n):
  if n > 0:
    print
    nLinhas(n-1)
```

Esse programa é similar a `contagemRegressiva`. Sempre que n for maior que 0, ele gera como saída uma nova linha e então chama a si mesmo para gerar como saída `n-1` linhas adicionais. Deste modo, o número total de novas linhas é `1 + (n-1)` que, se você estudou álgebra direitinho, vem a ser o próprio `n`.

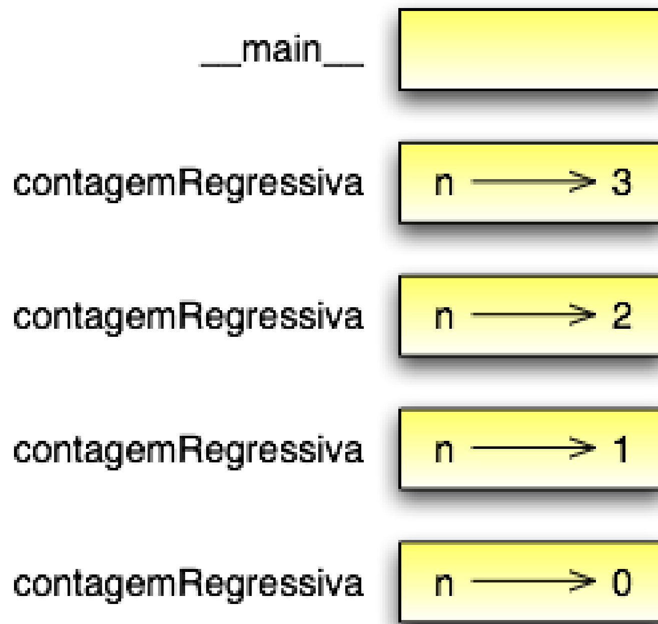
O processo de uma função chamando a si mesma é chamado de **recursividade**, e tais funções são ditas recursivas.

6.10 4.10 Diagramas de pilha para funções recursivas

Na Seção 3.11, usamos um diagrama de pilha para representar o estado de um programa durante uma chamada de função. O mesmo tipo de diagrama pode ajudar a interpretar uma função recursiva.

Toda vez que uma função é chamada, Python cria um novo quadro (*frame*) para a função, que contém as variáveis locais e parâmetros da função. Para uma função recursiva, terá que existir mais de um quadro na pilha ao mesmo tempo.

Esta figura mostra um diagrama de pilha para `contagemRegressiva`, chamada com `n = 3`:



Como de costume, no topo da pilha está o quadro para `__main__`. Ele está vazio porque nem criamos qualquer variável em `__main__` nem passamos qualquer valor para ele.

Os quatro quadros `contagemRegressiva` têm valores diferentes para o parâmetro `n`. A parte mais em baixo na pilha, onde `n=0`, é chamada de **caso base**. Ele não faz uma chamada recursiva, então não há mais quadros.

> Como exercício, desenhe um diagrama de pilha para `nLinhas` chamada com `n=4`.

6.11 4.11 Recursividade infinita

Se uma recursividade nunca chega ao caso base, ela prossegue fazendo chamadas recursivas para sempre, e o programa nunca termina. Isto é conhecido como recursividade infinita, e geralmente não é considerada uma boa ideia. Aqui está um programa mínimo com uma recursividade infinita:

```
def recursiva():
  recursiva()
```

Na maioria dos ambientes de programação, um programa com recursividade infinita na verdade não roda para sempre. Python reporta uma mensagem de erro quando a profundidade máxima de recursividade é alcançada:

```
File "<stdin>", line 2, in recursiva
(98 repetitions omitted)
File "<stdin>", line 2, in recursiva
RuntimeError: Maximum recursion depth exceeded
```

Este traceback é um pouco maior do que aquele que vimos no capítulo anterior. Quando o erro ocorre, existem 100 quadros `recursiva` na pilha!

> Como exercício, escreva uma função com recursividade infinita e rode-a no interpretador Python.

6.12 4.12 Entrada pelo teclado

Os programas que temos escrito até agora são um pouco crus, no sentido de não aceitarem dados entrados pelo usuário. Eles simplesmente fazem a mesma coisa todas as vezes.

Python fornece funções nativas que pegam entradas pelo teclado. A mais simples é chamada `raw_input`. Quando esta função é chamada, o programa pára e espera que o usuário digite alguma coisa. Quando o usuário aperta a tecla Enter ou Return, o programa prossegue e a função `raw_input` retorna o que o usuário digitou como uma `string`:

```
>>> entrada = raw_input()
O que você está esperando?
>>> print entrada
O que você está esperando?
```

Antes de chamar `raw_input`, é uma boa ideia exibir uma mensagem dizendo ao usuário o que ele deve entrar. Esta mensagem é uma como se fosse uma pergunta (*prompt*). Esta pergunta pode ser enviada como um argumento para `raw_input`:

```
>>> nome = raw_input("Qual... é o seu nome? ")
Qual... é o seu nome? Arthur, Rei dos Bretões!
>>> print nome
Arthur, Rei dos Bretões!
```

Se esperamos que a entrada seja um inteiro, podemos usar a função `input`:

```
pergunta = "Qual... é a velocidade de vôo de uma andorinha?\n"
velocidade = input(pergunta)
```

Se o usuário digita uma string de números, ela é convertida para um inteiro e atribuída a `velocidade`. Infelizmente, se o usuário digitar um caractere que não seja um número, o programa trava:

```
>>> velocidade = input(pergunta)
Qual... é a velocidade de vôo de uma andorinha?
De qual você fala, uma andorinha Africana ou uma Europeia?
SyntaxError: invalid syntax
```

Para evitar esse tipo de erro, geralmente é bom usar `raw_input` para pegar uma string e, então, usar funções de conversão para converter para outros tipos.

6.13 4.13 Glossário

aninhamento (*nesting*) Estrutura de programa dentro da outra, como um comando condicional dentro de um bloco de outro comando condicional.

bloco (*block*) Grupo de comandos consecutivos com a mesma endentação.

caso base (*base case*) Bloco de comando condicional numa função recursiva que não resulta em uma chamada recursiva.

comando composto (*compound statement*) Comando que consiste de um cabeçalho e um corpo. O cabeçalho termina com um dois-pontos (:). O corpo é endentado em relação ao cabeçalho.

comando condicional (*conditional statement*) Comando que controla o fluxo de execução dependendo de alguma condição.

condição (*condition*) A expressão booleana que determina qual bloco será executado num comando condicional.

corpo (*body*) O bloco que se segue ao cabeçalho em um comando composto.

expressão booleana (*boolean expression*) Uma expressão que é verdadeira ou falsa.

operador de comparação (*comparison operator*) Um dos operadores que compara dois valores: `==`, `!=`, `>`, `<`, `>=`, e `<=`.

operador lógico (*logical operator*) Um dos operadores que combina expressões booleanas: `and`, `or`, e `not`.

operador módulo (*modulus operator*) Operador denotado por um símbolo de porcentagem (%), que trabalha com inteiros e retorna o resto da divisão de um número por outro.

prompt Indicação visual que diz ao usuário que o programa está esperando uma entrada de dados.

recursividade (*recursion*) O processo de chamar a própria função que está sendo executada.

recursividade infinita (*infinite recursion*) Função que chama a si mesma recursivamente sem nunca chegar ao caso base. Após algum tempo, uma recursividade infinita causa um erro de execução.

Capítulo 5: Funções frutíferas

Tópicos

7.1 5.1 Valores de retorno

Algumas das funções nativas do Python que temos usado, como as funções matemáticas, produziram resultados. Chamar a função gerou um novo valor, o qual geralmente atribuímos à uma variável ou usamos como parte de uma expressão:

```
e = math.exp(1.0)
altura = raio * math.sin(angulo)
```

Mas até agora, nenhuma das funções que nós escrevemos retornou um valor.

Neste capítulo, iremos escrever funções que retornam valores, as quais chamaremos de **funções frutíferas**, ou funções que dão frutos, na falta de um nome melhor. O primeiro exemplo é `area`, que retorna a área de um círculo dado o seu raio:

```
import math

def area(raio):
  temp = math.pi * raio**2
  return temp
```

Já vimos a instrução `return` antes, mas em uma função frutífera a instrução `return` inclui um **valor de retorno**. Esta instrução significa: "Retorne imediatamente desta função e use a expressão em seguida como um valor de retorno". A expressão fornecida pode ser arbitrariamente complicada, de modo que poderíamos ter escrito esta função de maneira mais concisa:

```
def area(raio):
  return math.pi * raio**2
```

Por outro lado, variáveis temporárias como `temp` muitas vezes tornam a depuração mais fácil.

Às vezes é útil ter múltiplos comandos `return`, um em cada ramo de uma condicional:

```
def valorAbsoluto(x):
  if x < 0:
    return -x
  else:
    return x
```

Já que estes comandos `return` estão em ramos alternativos da condicional, apenas um será executado. Tão logo um seja executado, a função termina sem executar qualquer instrução ou comando subsequente.

O código que aparece depois de uma instrução `return`, ou em qualquer outro lugar que o fluxo de execução jamais alcance, é chamado código morto (*dead code*).

Em uma função frutífera, é uma boa ideia assegurar que todo caminho possível dentro do programa encontre uma instrução `return`. Por exemplo:

```
def valorAbsoluto(x):
  if x < 0:
    return -x
  elif x > 0:
    return x
```

Este programa não está correto porque se `x` for 0, nenhuma das condições será verdadeira, e a função terminará sem encontrar um comando `return`. Neste caso, o valor de retorno será um valor especial chamado `None`:

```
>>> print valorAbsoluto(0)
None
```

Como exercício, escreva uma função compare que retorne 1 se x > y, 0 se x == y e -1 se x < y.

7.2 5.2 Desenvolvimento de programas

Neste ponto, você deve estar apto a olhar para funções completas e dizer o que elas fazem. Também, se você vem fazendo os exercícios, você escreveu algumas pequenas funções. Conforme escrever funções maiores, você pode começar a ter mais dificuldade, especialmente com erros em tempo de execução (erros de runtime) ou erros semânticos.

Para lidar com programas de crescente complexidade, vamos sugerir uma técnica chamada desenvolvimento incremental. A meta do desenvolvimento incremental é evitar seções de depuração (*debugging*) muito longas pela adição e teste de somente uma pequena quantidade de código de cada vez.

Como exemplo, suponha que você queira encontrar a distância entre dois pontos, dados pelas coordenadas (x1,y1) e (x2,y2). Pelo teorema de Pitágoras, a distância é:

```
distancia = V (x2 - x1)2 + (y2 - y1)2          (5.1)
```

XXX: falta o sinal de raiz e elevar os expoentes desta fórmula

O primeiro passo é considerar como deveria ser uma função `distancia` em Python. Em outras palavras, quais são as entradas (parâmetros) e qual é a saída (valor de retorno)?

Neste caso, os dois pontos são as entradas, os quais podemos representar usando quatro parâmetros. O valor de retorno é a distância, que é um valor em ponto flutuante.

Já podemos escrever um esboço da função:

```
def distancia(x1, y1, x2, y2):
  return 0.0
```

Obviamente, esta versão da função não calcula distâncias; ela sempre retorna zero. Mas ela está sintaticamente correta, e vai rodar, o que significa que podemos testá-la antes de torná-la mais complicada.

Para testar a nova função, vamos chamá-la com valores hipotéticos:

```
>>> distancia(1, 2, 4, 6)
0.0
```

Escolhemos estes valores de modo que a distância horizontal seja igual a 3 e a distância vertical seja igual a 4; deste modo, o resultado é 5 (a hipotenusa de um triângulo 3-4-5). Quando testamos uma função, é útil sabermos qual o resultado correto.

Neste ponto, já confirmamos que a função está sintaticamente correta, e podemos começar a adicionar linhas de código. Depois de cada mudança adicionada, testamos a função de novo. Se um erro ocorre em qualquer ponto, sabemos aonde ele deve estar: nas linhas adicionadas mais recentemente.

Um primeiro passo lógico nesta operação é encontrar as diferenças x2 - x1 e y2 - y1. Nós iremos guardar estes valores em variáveis temporárias chamadas `dx` e `dy` e imprimi-las:

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  print "dx vale", dx
  print "dy vale", dy
  return 0.0
```

Se a função estiver funcionando, as saídas deverão ser 3 e 4. Se é assim, sabemos que a função está recebendo os parâmetros corretos e realizando o primeiro cálculo corretamente. Se não, existem poucas linhas para checar.

Em seguida, calcularemos a soma dos quadrados de `dx` e `dy`:

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  dquadrado = dx**2 + dy**2
  print "dquadrado vale: ", dquadrado
  return 0.0
```

Note que removemos os comandos `print` que havíamos escrito no passo anterior. Código como este ajuda a escrever o programa, mas não é parte do produto final (em inglês é usado o termo *scaffolding*).

De novo, nós vamos rodar o programa neste estágio e checar a saída (que deveria ser 25).

Finalmente, se nós tínhamos importado o módulo matemático `math`, podemos usar a função `sqrt` para computar e retornar o resultado:

```
def distancia(x1, x2, y1, y2):
  dx = x2 - x1
  dy = y2 - y1
  dquadrado = dx**2 + dy**2
  resultado = math.sqrt(dquadrado)
  return resultado
```

Se isto funcionar corretamente, você conseguiu. Caso contrário, talvez fosse preciso imprimir (exibir) o valor de `resultado` antes da instrução `return`.

Enquanto for iniciante, você deve acrescentar apenas uma ou duas linhas de código de cada vez. Conforme ganhar mais experiência, você se verá escrevendo e depurando pedaços maiores. De qualquer modo, o processo de desenvolvimento incremental pode poupar um bocado de tempo de depuração.

Os aspectos chave do processo são:

1. Comece com um programa que funciona e faça pequenas mudanças incrementais. Em qualquer ponto do processo, se houver um erro, você saberá exatamente onde ele está.
2. Use variáveis temporárias para manter valores intermediários de modo que você possa exibi-los e checá-los.
3. Uma vez que o programa funcione, você pode querer remover algum código muleta, ou algum *scaffolding* ou consolidar múltiplos comandos dentro de expressões compostas, mas somente se isto não tornar o programa difícil de ler.

   Como um exercício, use o desenvolvimento incremental para escrever uma função chamada `hipotenusa` que retorna a medida da hipotenusa de um triângulo retângulo dadas as medidas dos dois catetos como parâmetros. Registre cada estágio do desenvolvimento incremental conforme você avance.

7.3 5.3 Composição

Conforme você poderia esperar agora, você pode chamar uma função de dentro de outra. Esta habilidade é chamada de **composição**.

Como um exemplo, vamos escrever uma função que recebe dois pontos, o centro de um círculo e um ponto em seu perímetro, e calcula a área do círculo.

Assuma que o ponto do centro está guardado nas variáveis `xc` e `yc`, e que o ponto do perímetro está nas variáveis `xp` e `yp`. O primeiro passo é encontrar o raio do círculo, o qual é a entre os dois pontos. Felizmente, temos uma função, `distancia`, que faz isto:

```
Raio = distancia(xc, yc, xp, yp)
```

O segundo passo é encontrar a área de um círculo com o raio dado e retorná-la:

```
resultado = area(raio)
return resultado
```

Juntando tudo numa função, temos:

```
def area2(xc, yc, xp, yp):

  raio = distancia(xc, yc, xp, yp)
  resultado = area(raio)
  return resultado
```

Chamamos à esta função de `area2` para distinguir da função `area`, definida anteriormente. Só pode existir uma única função com um determinado nome em um determinado módulo.

As variáveis temporárias `raio` e `resultado` são úteis para o desenvolvimento e para depuração (*debugging*), mas uma vez que o programa esteja funcionando, podemos torná-lo mais conciso através da composição das chamadas de função:

```
def area2(xc, yc, xp, yp):

  return area(distancia(xc, yc, xp, yp))
```

Como exercício, escreva uma função `coeficienteAngular(x1, y1, x2, y2)` que retorne a coeficiente angular de uma linha dados os pontos (x1, y1) e (x2, y2). Depois use esta função em uma função chamada `cortaY(x1, y1, x2, y2)` que retorne a interseção da linha com o eixo y, dados os pontos (x1, y1) e (x2, y2).

7.4 5.4 Funções booleanas

Funções podem retornar valores booleanos, o que muitas vezes é conveniente por ocultar testes complicados dentro de funções. Por exemplo:

```
def ehDivisivel(x, y):
  If  x % y == 0:
    return True  # é verdadeiro (True), é divisível
  else:
    return False # é falso (False), não é divisível
```

O nome desta função é `ehDivisivel` ("é divisível"). É comum dar a uma função booleana nomes que soem como perguntas sim/não. `ehDivisivel` retorna ou `True` ou `False` para indicar se `x` é ou não é divisível por `y`.

Podemos tornar a função mais concisa se tirarmos vantagem do fato de a condição da instrução `if` ser ela mesma uma expressão booleana. Podemos retorná-la diretamente, evitando totalmente o `if`:

```
def ehDivisivel(x, y):
  return x % y == 0
```

Esta sessão mostra a nova função em ação:

```
>>> ehDivisivel(6, 4)
False
>>> ehDivisivel(6, 3)
True
```

Funções booleanas são frequentemente usadas em comandos condicionais:

```
if ehDivisivel(x, y):
  print "x é divisível por y"
else:
  print "x não é divisível por y"
```

Mas a comparação extra é desnecessária.

> Como exercício, escreva uma função `estaEntre(x, y, z)` que retorne `True` se $y < x < z$ ou `False`, se não.

7.5 5.5 Mais recursividade

Até aqui, você aprendeu apenas um pequeno subconjunto da linguagem Python, mas pode ser que te interesse saber que este pequeno subconjunto é uma linguagem de programação completa, o que significa que qualquer coisa que possa ser traduzida em operação computacional pode ser expressa nesta linguagem. Qualquer programa já escrito pode ser reescrito usando somente os aspectos da linguagem que você aprendeu até agora (usualmente, você precisaria de uns poucos comandos para controlar dispositivos como o teclado, mouse, discos, etc., mas isto é tudo).

Provar esta afirmação é um exercício nada trivial, que foi alcançado pela primeira vez por Alan Turing, um dos primeiros cientistas da computação (alguém poderia dizer que ele foi um matemático, mas muitos dos primeiros cientistas da computação começaram como matemáticos). Por isso, ficou conhecido como Tese de Turing. Se você fizer um curso em Teoria da Computação, você terá chance de ver a prova.

Para te dar uma ideia do que você pode fazer com as ferramentas que aprendeu a usar até agora, vamos avaliar algumas funções matemáticas recursivamente definidas. Uma definição recursiva é similar à uma definição circular, no sentido de que a definição faz referência à coisa que está sendo definida. Uma verdadeira definição circular não é muito útil:

vorpal: adjetivo usado para descrever algo que é vorpal.

Se você visse esta definição em um dicionário, ficaria confuso. Por outro lado, se você procurasse pela definição da função matemática fatorial, você encontraria algo assim:

```
0! = 1
n! = n.(n-1)!
```

Esta definição diz que o fatorial de 0 é 1, e que o fatorial de qualquer outro valor, n, é n multiplicado pelo fatorial de n-1.

Assim, 3! (lê-se "3 fatorial" ou "fatorial de 3") é 3 vezes 2!, o qual é 2 vezes 1!, o qual é 1 vezes 0!. Colocando tudo isso junto, 3! igual 3 vezes 2 vezes 1 vezes 1, o que é 6.

Se você pode escrever uma definição recursiva de alguma coisa, você geralmente pode escrever um programa em Python para executá-la. O primeiro passo é decidir quais são os parâmetros para esta função. Com pouco esforço, você deverá concluir que `fatorial` recebe um único parâmetro:

```
def fatorial(n):
```

Se acontece de o argumento ser 0, tudo o que temos de fazer é retornar 1:

```
def fatorial(n):
  if n == 0:
    return 1
```

Por outro lado, e esta é a parte interessante, temos que fazer uma chamada recursiva para encontrar o fatorial de n-1 e então multiplicá-lo por n:

```
def fatorial(n):
  if n == 0:
    return 1
  else:
    recursivo = fatorial(n-1)
    resultado = n * recursivo
    return resultado
```

O fluxo de execução para este programa é similar ao fluxo de `contagemRegressiva` na Seção 4.9. Se chamarmos `fatorial` com o valor 3:

Já que 3 não é 0, tomamos o segundo ramo e calculamos o fatorial de `n-1` ...

Já que 2 não é 0, tomamos o segundo ramo e calculamos o fatorial de `n-1` ...

Já que 1 não é 0, tomamos o segundo ramo e calculamos o fatorial de `n-1` ...

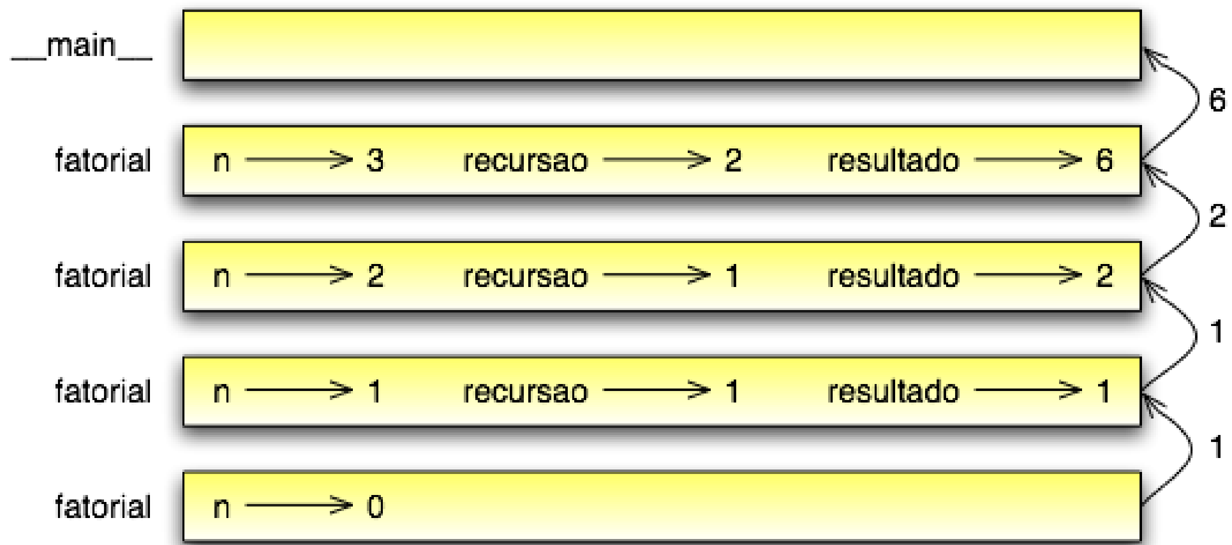
Já que 0 *é* 0, tomamos o primeiro ramo e retornamos 1 sem fazer mais qualquer chamada recursiva.

O valor retornado (1) é multiplicado por `n`, que é 1, e o resultado é retornado.

O valor retornado (1) é multiplicado por `n`, que é 2, e o resultado é retornado.

O valor retornado (2) é multiplicado por `n`, que é 3, e o resultado, 6, se torna o valor de retorno da chamada de função que iniciou todo o processo.

Eis o diagrama de pilha para esta sequência de chamadas de função:



Os valores de retorno são mostrados sendo passados de volta para cima da pilha. Em cada quadro, o valor de retorno é o valor de `resultado`, o qual é o produto de `n` por `recursivo`.

7.6 5.6 Voto de confiança (Leap of faith)

Seguir o fluxo de execução é uma maneira de ler programas, mas que pode rapidamente se transformar em um labirinto. Uma alternativa é o que chamamos de "voto de confiança". Quando você tem uma chamada de função, em vez de seguir o fluxo de execução, você *assume* que a função funciona corretamente e retorna o valor apropriado.

De fato, você está agora mesmo praticando este voto de confiança ao usar as funções nativas. Quando você chama `math.cos` ou `math.exp`, você não examina a implementação destas funções. Você apenas assume que elas funcionam porque as pessoas que escreveram as bibliotecas nativas eram bons programadores.

O mesmo também é verdade quando você chama uma de suas próprias funções. Por exemplo, na Seção 5.4, escrevemos a função chamada `ehDivisivel` que determina se um número é divisível por outro. Uma vez que nos convencemos que esta função está correta – ao testar e examinar o código – podemos usar a função sem examinar o código novamente.

O mesmo também é verdadeiro para programas recursivos. Quando você tem uma chamada recursiva, em vez de seguir o fluxo de execução, você poderia assumir que a chamada recursiva funciona (produz o resultado correto) e então perguntar-se, "Assumindo que eu possa encontrar o fatorial de *n-1*, posso calcular o fatorial de *n*?" Neste caso, é claro que você pode, multiplicando por *n*.

Naturalmente, é um pouco estranho que uma função funcione corretamente se você ainda nem terminou de escrevê-la, mas é por isso que se chama voto de confiança!

7.7 5.7 Mais um exemplo

No exemplo anterior, usamos variáveis temporárias para deixar claros os passos e tornar o código mais fácil de depurar, mas poderíamos ter economizado algumas linhas:

```
def fatorial(n):
  if n == 0:
    return 1
```

```
  else:
    return n * fatorial(n-1)
```

De agora em diante, tenderemos a utilizar um formato mais conciso, mas recomendamos que você use a versão mais explícita enquanto estiver desenvolvendo código. Quando ele estiver funcionando, você pode enxugá-lo se estiver se sentindo inspirado.

Depois de `fatorial`, o exemplo mais comum de uma função matemática definida recursivamente é `fibonacci`, a qual tem a seguinte definição:

```
fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
```

Traduzido em Python, parecerá assim:

```
def fibonacci(n):
  if n == 0 or n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

Se você tentar seguir o fluxo de execução aqui, mesmo para valores bem pequenos de n, sua cabeça explodirá. Mas, de acordo com o voto de confiança, se você assume que as duas chamadas recursivas funcionam corretamente, então é claro que você terá o resultado correto ao juntá-las.

7.8 5.8 Checagem de tipos

O que acontece se chamamos `fatorial` e damos a ela 1.5 como argumento?:

```
>>> fatorial (1.5)
RuntimeError: Maximum recursion depth exceeded
```

Parece um caso de recursividade infinita. Mas o que será que é de fato? Existe um caso base – quando `n == 0`. O problema é que o valor de `n` *nunca encontra* o caso base.

Na primeira chamada recursiva, o valor de `n` é 0.5. Na próxima, ele é igual a -0.5. Daí em diante, ele se torna cada vez menor, mas jamais será 0.

Temos então duas alternativas. Podemos tentar generalizar a função `fatorial` para que funcione com números em ponto flutuante, ou fazemos `fatorial` realizar a checagem de tipo de seus parâmetros. A primeira é chamada função gamma e está um pouco além do escopo deste livro. Sendo assim, ficaremos com a segunda.

Podemos usar `type` para comparar o tipo do parâmetro com o tipo de um valor inteiro conhecido (como 1). Ao mesmo tempo em que fazemos isto, podemos nos certificar também de que o parâmetro seja positivo:

```
def fatorial (n):
  if type(n) != type(1):
    print "Fatorial somente é definido para inteiros."
    return -1
  elif n < 0:
    print "Fatorial somente é definido para inteiros positivos."
    return -1
  elif n ==0:
    return 1
  else:
    return n * fatorial(n-1)
```

Agora temos três casos base. O primeiro pega os não-inteiros. O segundo pega os inteiros negativos. Em ambos os casos, o programa exibe uma mensagem de erro e retorna um valor especial, -1, para indicar que alguma coisa saiu errada:

```
>>> fatorial ("Fred")
Fatorial somente é definido para inteiros.
-1
>>> fatorial (-2)
Fatorial somente é definido para inteiros positivos.
-1
```

Se passarmos pelas duas checagens, então saberemos que `n` é um inteiro positivo, e poderemos provar que a recursividade encontra seu término.

Este programa demonstra um padrão (*pattern*) chamado às vezes de **guardião**. As duas primeiras condicionais atuam como guardiãs, protegendo o código que vem em seguida de valores que poderiam causar um erro. Os guardiões tornam possível garantir a correção do código.

7.9 5.9 Glossário

código morto (*dead code*) Parte de um programa que nunca pode ser executada, muitas vezes por que ela aparece depois de uma instrução `return`.

desenvolvimento incremental (*incremental development*) Uma estratégia de desenvolvimento de programas que evita a depuração ao adicionar e testar somente uma pequena quantidade de código de cada vez.

função frutífera (*fruitful function*) Uma função que produz um valor de retorno.

guardião (*guardian*) Uma condição que checa e manipula circunstâncias que poderiam causar um erro.

`None` Um valor especial em Python, retornado por funções que não possuem uma instrução `return` ou têm uma instrução `return` sem argumento.

scaffolding Código usado durante o desenvolvimento do programa, mas que não faz parte do produto final.

variável temporária (*temporary variable*) Uma variável usada para guardar um valor intermediário em um cálculo complexo.

valor de retorno (*return value*) O valor entregue como resultado de uma chamada de função.

Capítulo 6: Iteração

Tópicos

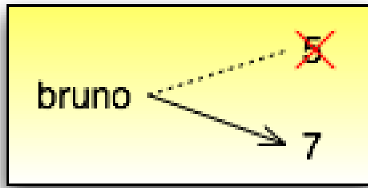
8.1 6.1 Reatribuições

Como você talvez já tenha descoberto, é permitido fazer mais de uma atribuição à mesma variável. Uma nova atribuição faz uma variável existente referir-se a um novo valor (sem se referir mais ao antigo).:

```
bruno = 5
print bruno,
bruno = 7
print bruno
```

A saída deste programa é `5 7`, porque na primeira vez que `bruno` é impresso, seu valor é 5 e na segunda vez, seu valor é 7. A vírgula no final do primeiro comando `print` suprime a nova linha no final da saída, que é o motivo pelo qual as duas saídas aparecem na mesma linha.

Veja uma **reatribuição** em um diagrama de estado:



Com a reatribuição torna-se ainda mais importante distinguir entre uma operação de atribuição e um comando de igualdade. Como Python usa o sinal de igual (=) para atribuição, existe a tendência de lermos um comando como `a = b` como um comando de igualdade. Mas não é!

Em primeiro lugar, igualdade é comutativa e atribuição não é. Por exemplo, em matemática, se a = 7 então 7 = a. Mas em Python, o comando `a = 7` é permitido e `7 = a` não é.

Além disso, em matemática, uma expressão de igualdade é sempre verdadeira. Se a = b agora, então, a será sempre igual a b. Em Python, um comando de atribuição pode tornar duas variáveis iguais, mas elas não têm que permanecer assim:

```
a = 5
b = a # a e b agora são iguais
b = 3 # a e b não são mais iguais
```

A terceira linha muda o valor de `a` mas não muda o valor de `b`, então, elas não são mais iguais. (Em algumas linguagens de programação, um símbolo diferente é usado para atribuição, como `<-` ou `:=`, para evitar confusão.)

Embora a reatribuição seja freqüentemente útil, você deve usá-la com cautela. Se o valor das variáveis muda freqüentemente, isto pode fazer o código difícil de ler e de depurar.

8.2 6.2 O comando `while`

Os computadores são muito utilizados para automatizar tarefas repetitivas. Repetir tarefas idênticas ou similares sem cometer erros é uma coisa que os computadores fazem bem e que as pessoas fazem poorly.

Vimos dois programas, `nLinhas` e `contagemRegressiva`, que usam recursividade (recursão) para fazer a repetição, que também é chamada **iteração**. Porque a iteração é muito comum, Python tem várias características para torná-la mais fácil. A primeira delas em que vamos dar uma olhada é o comando `while`.

Aqui está como fica `contagemRegressiva` com um comando `while`:

```
def contagemRegressiva(n):
  while n > 0:
    print n
    n = n-1
  print "Fogo!"
```

Desde que removemos a chamada recursiva, esta função não é recursiva.

Você quase pode ler o comando `while` como se fosse Inglês. Ele significa, "Enquanto (while) `n` for maior do que `0`, siga exibindo o valor de `n` e diminuindo `1` do valor de `n`. Quando chegar a `0`, exiba a palavra `Fogo!`".

Mais formalmente, aqui está o fluxo de execução para um comando `while`:

1. Teste a condição, resultando 0 ou 1.
2. Se a condição for falsa (0), saia do comando while e continue a execução a partir do próximo comando.
3. Se a condição for verdadeira (1), execute cada um dos comandos dentro do corpo e volte ao passo 1.

O corpo consiste de todos os comandos abaixo do cabeçalho, com a mesma endentação.

Este tipo de fluxo é chamado de um **loop** (ou laço) porque o terceiro passo cria um "loop" ou um laço de volta ao topo. Note que se a condição for falsa na primeira vez que entrarmos no loop, os comandos dentro do loop jamais serão executados.

O corpo do loop poderia alterar o valor de uma ou mais variáveis de modo que eventualmente a condição se torne falsa e o loop termine. Se não for assim, o loop se repetirá para sempre, o que é chamado de um **loop infinito**. Uma fonte de diversão sem fim para os cientistas da computação é a observação de que as instruções da embalagem de shampoo, "Lave, enxágüe, repita" é um loop infinito.

No caso de `contagemRegressiva`, podemos provar que o loop terminará porque sabemos que o valor de n é finito, e podemos ver que o valor de n diminui dentro de cada repetição (iteração) do loop, então, eventualmente chegaremos ao 0. Em outros casos, isto não é tão simples de afirmar:

```
def sequencia(n):
  while n != 1:
    print n,
    if n%2 == 0:            # n é par
      n = n/2
    else:                   # n é impar
      n = n*3+1
```

A condição para este loop é `n != 1`, então o loop vai continuar até que `n` seja `1`, o que tornará a condição falsa.

Dentro de cada repetição (iteração) do loop, o programa gera o valor de `n` e então checa se ele é par ou impar. Se ele for par, o valor de `n` é dividido por `2`. Se ele for impar, o valor é substituído por `n*3+1`. Por exemplo, se o valor inicial (o argumento passado para `seqüência`) for 3, a seqüência resultante será `3`, `10`, `5`, `16`, `8`, `4`, `2`, `1`.

Já que `n` às vezes aumenta e às vezes diminui, não existe uma prova óbvia de que `n` jamais venha a alcançar 1, ou de que o programa termine. Para alguns valores particulares de `n`, podemos provar o término. Por exemplo, se o valor inicial for uma potência de dois, então o valor de `n` será par dentro de cada repetição (iteração) do loop até que alcance 1. O exemplo anterior termina com uma dessas seqüências começando em 16.

Valores específicos à parte, A questão interessante é se há como provarmos que este programa termina para todos os valores de `n`. Até hoje, ninguém foi capaz de provar que sim ou que não!

Como um exercício, reescreva a função `nLinhas` da seção 4.9 usando iteração em vez de recursão.

8.3 6.3 Tabelas

Uma das coisas para qual os loops são bons é para gerar dados tabulares. Antes que os computadores estivessem readily disponíveis, as pessoas tinham que calcular logaritmos, senos, cossenos e outras funções matemáticas à mão. Para tornar isto mais fácil, os livros de matemática continham longas tabelas listando os valores destas funções. Criar as tabelas era demorado e entediante, e elas tendiam a ser cheias de erros.

Quando os computadores entraram em cena, uma das reações iniciais foi "Isto é ótimo! Podemos usar computadores para geras as tabelas, assim não haverá erros." Isto veio a se tornar verdade (na maioria das vezes) mas shortsighted. Rapidamente, porém, computadores e calculadoras tornaram-se tão pervasivos que as tabelas ficaram obsoletas.

Bem, quase. Para algumas operações, os computadores usam tabelas de valores para conseguir uma resposta aproximada e então realizar cálculos para melhorar a aproximação. Em alguns casos, têm havido erros nas tabelas underlying, o caso mais famoso sendo o da tabela usada pelo processador Pentium da Intel para executar a divisão em ponto-flutuante.

Embora uma tabela de logaritmos não seja mais tão útil quanto já foi um dia, ela ainda dá um bom exemplo de iteração. O seguinte programa gera uma seqüência de valores na coluna da esquerda e seus respectivos logaritmos na coluna da direita:

```
x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

A string `'\t'` representa um caracter de **tabulação**.

Conforme caracteres e strings vão sendo mostrados na tela, um ponteiro invisível chamado **cursor** marca aonde aparecerá o próximo caractere. Depois de um comando `print`, o cursor normalmente vai para o início de uma nova linha.

O caractere de tabulação desloca o cursor para a direita até que ele encontre uma das marcas de tabulação. Tabulação é útil para fazer colunas de texto line up, como na saída do programa anterior:

```
1.0   0.0
2.0   0.69314718056
3.0   1.09861228867
4.0   1.38629436112
5.0   1.60943791243
6.0   1.79175946923
7.0   1.94591014906
8.0   2.07944154168
9.0   2.19722457734
```

Se estes valores parecem odd, lembre-se que a função `log` usa a base `e`. Já que potências de dois são tão importantes em ciência da computação, nós freqüentemente temos que achar logaritmos referentes à base 2. Para fazermos isso, podemos usar a seguinte fórmula:

(XXX diagramar fórmula matemática)

log2 x = loge x (6.1) loge 2

Alterando o comando de saída para:

```
print x, '\t', math.log(x)/math.log(2.0)
```

o que resultará em:

```
1.0   0.0
2.0   1.0
3.0   1.58496250072
4.0   2.0
5.0   2.32192809489
6.0   2.58496250072
7.0   2.80735492206
8.0   3.0
9.0   3.16992500144
```

Podemos ver que 1, 2, 4 e 8 são potências de dois porque seus logaritmos na base 2 são números redondos. Se precisássemos encontrar os logaritmos de outras potências de dois, poderíamos modificar o programa deste modo:

```
x = 1.0
while x < 100.0:
  print x, '\t', math.log(x)/math.log(2.0)
  x = x * 2.0
```

Agora, em vez de somar algo a x a cada iteração do loop, o que resulta numa seqüência aritmética, nós multiplicamos x por algo, resultando numa seqüência geométrica. O resultado é:

```
1.0   0.0
2.0   1.0
```

```
4.0    2.0
8.0    3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

Por causa do caractere de tabulação entre as colunas, a posição da segunda coluna não depende do número de dígitos na primeira coluna.

Tabelas de logaritmos podem não ser mais úteis, mas para cientistas da computação, conhecer as potências de dois é!

Como um exercício, modifique este programa de modo que ele produza as potências de dois acima de 65.535 (ou seja, 216). Imprima e memorize-as.

O caractere de barra invertida em `'\t'` indica o início de uma seqüência de escape. Seqüências de escape são usadas para representar caracteres invisíveis como de tabulação e de nova linha. A seqüência `\n` representa uma nova linha.

Uma seqüência de escape pode aparecer em qualquer lugar em uma string; no exemplo, a seqüência de escape de tabulação é a única coisa dentro da string.

Como você acha que se representa uma barra invertida em uma string?

Como um exercício, escreva um única string que

produza

esta saída.

8.4 6.4 Tabelas de duas dimensões (ou bi-dimensionais)

Uma tabela de duas dimensões é uma tabela em que você lê o valor na interseção entre uma linha e uma coluna. Uma tabela de multiplicação é um bom exemplo. Digamos que você queira imprimir uma tabela de multiplicação de 1 a 6.

Uma boa maneira de começar é escrever um loop que imprima os múltiplos de 2, todos em uma linha:

```
i = 1
while i <= 6:
  print 2*i, '   ',
  i = i + 1
print
```

A primeira linha inicializa a variável chamada `i`, a qual age como um contador ou **variável de controle do loop**. Conforme o loop é executado, o valor de `i` é incrementado de 1 a 6. Quando `i` for 7, o loop termina. A cada repetição (iteração) do loop, é mostrado o valor de `2*i`, seguido de três espaços.

De novo, a vírgula no comando `print` suprime a nova linha. Depois que o loop se completa, o segundo comando `print` inicia uma nova linha.

A saída do programa é:

```
2      4      6      8      10      12
```

Até aqui, tudo bem. O próximo passo é **encapsular** e **generalizar**.

8.5 6.5 Encapsulamento e generalização

Encapsulamento é o processo de wrapping um pedaço de código em uma função, permitindo que você tire vantagem de todas as coisas para as quais as funções são boas. Você já viu dois exemplos de encapsulamento: `imprimeParidade`

na seção 4.5; e `eDivisivel` na seção 5.4

Generalização significa tomar algo que é específico, tal como imprimir os múltiplos de 2, e torná-lo mais geral, tal como imprimir os múltiplos de qualquer inteiro.

Esta função encapsula o loop anterior e generaliza-o para imprimir múltiplos de n:

```
def imprimeMultiplos(n):
i = 1
while i <= 6:
  print n*i, '\t ',
  i = i + 1
print
```

Para encapsular, tudo o que tivemos que fazer foi adicionar a primeira linha, que declara o nome de uma função e sua lista de parâmetros. Para generalizar, tudo o que tivemos que fazer foi substituir o valor 2 pelo parâmetro `n`.

Se chamarmos esta função com o argumento 2, teremos a mesma saída que antes. Com o argumento `3`, a saída é:

```
3      6       9       12      15      18
```

Com o argumento `4`, a saída é:

```
4      8       12      16      20      24
```

Agora você provavelmente pode adivinhar como imprimir uma tabela de multiplicação - chamando `imprimeMultiplos` repetidamente com argumentos diferentes. De fato, podemos usar um outro loop:

```
i = 1
while i <= 6:
  imprimeMultiplos(i)
  i = i + 1
```

Note o quanto este loop é parecido com aquele dentro de `imprimeMultiplos`. Tudo o que fiz foi substituir o comando `print` pela chamada à função.

A saída deste programa é uma tabela de multiplicação:

```
1      2       3       4       5       6
2      4       6       8       10      12
3      6       9       12      15      18
4      8       12      16      20      24
5      10      15      20      25      30
6      12      18      24      30      36
```

8.6 6.6 Mais encapsulamento

Para demonstrar de novo o encapsulamento, vamos pegar o código do final da seção 6.5 e acondicioná-lo, envolvê-lo em uma função:

```
def imprimeTabMult():
  i = 1
  while i <= 6:
    imprimeMultiplos(i)
    i = i + 1
```

Este processo é um **plano de desenvolvimento** comum. Nós desenvolvemos código escrevendo linhas de código fora de qualquer função, ou digitando-as no interpretador. Quando temos o código funcionando, extraímos ele e o embalamos em uma função.

Este plano de desenvolvimento é particularmente útil se você não sabe, quando você começa a escrever, como dividir o programa em funções. Esta técnica permite a você projetar enquanto desenvolve.

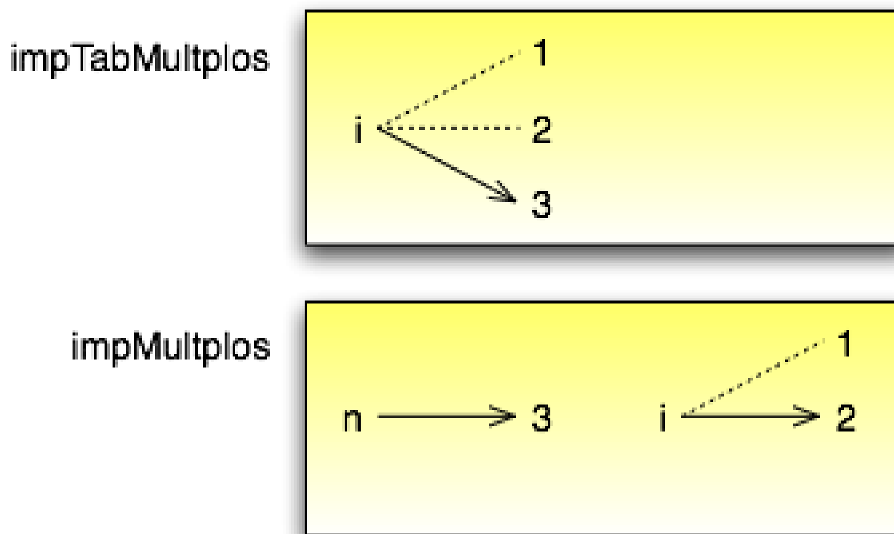
8.7 6.7 Variáveis locais

Você pode estar pensando como podemos utilizar a mesma variável, `i`, em ambos, `imprimeMultiplos` e `imprimeTabMult`. Isto não causaria problemas quando uma das funções mudasse o valor da variável?

A resposta é não, porque o `i` em `imprimeMultiplos` e o `i` em `imprimeTabMult` não são a mesma variável.

Variáveis criadas dentro de uma definição de função são locais; você não pode acessar uma variável local de fora da função em que ela "mora". Isto significa que você é livre para ter múltiplas variáveis com o mesmo nome, desde que elas não estejam dentro da mesma função.

O diagrama de pilha para este programa mostra que duas variáveis chamadas `i` não são a mesma variável. Elas podem se referir a valores diferentes e alterar o valor de uma não afeta à outra.



O valor de `i` em `imprimeTabMult` vai de 1 a 6. No diagrama, `i` agora é 3. Na próxima iteração do loop `i` será 4. A cada iteração do loop, `imprimeTabMult` chama `imprimeMultiplos` com o valor corrente de `i` como argumento. O valor é atribuído ao parâmetro `n`.

Dentro de `imprimeMultiplos`, o valor de `i` vai de 1 a 6. No diagrama, `i` agora é `2`. Mudar esta variável não tem efeito sobre o valor de `i` em `imprimeTabMult`.

É comum e perfeitamente legal ter variáveis locais diferentes com o mesmo nome. Em particular, nomes como `i` e `j` são muito usados para variáveis de controle de loop. Se você evitar utilizá-los em uma função só porque você já os usou em outro lugar, você provavelmente tornará seu programa mais difícil de ler.

8.8 6.8 Mais generalização

Como um outro exemplo de generalização, imagine que você precise de um programa que possa imprimir uma tabela de multiplicação de qualquer tamanho, não apenas uma tabela de seis por seis. Você poderia adicionar um parâmetro a `imprimeTabMult`:

```
def imprimeTabMult(altura):
  i = 1
  while i <= altura:
    imprimeMultiplos(i)
    i = i + 1
```

Nós substituímos o valor 6 pelo parâmetro altura. Se chamarmos `imprimeTabMult` com o argumento 7, ela mostra:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

Isto é bom, exceto que nós provavelmente quereríamos que a tabela fosse quadrada - com o mesmo número de linhas e colunas. Para fazer isso, adicionamos outro parâmetro a `imprimeMultiplos` para especificar quantas colunas a tabela deveria ter.

Só para confundir, chamamos este novo parâmetro de `altura`, demonstrando que diferentes funções podem ter parâmetros com o mesmo nome (como acontece com as variáveis locais). Aqui está o programa completo:

```
def imprimeMultiplos(n, altura):
  i = 1
  while i <= altura:
    print n*i, 't',
    i = i + 1
    print

def imprimeTabMult(altura):
  i = 1
  while i <= altura:
    imprimeMultiplos(i, altura)
    i = i + 1
```

Note que quando adicionamos um novo parâmetro, temos que mudar a primeira linha da função (o cabeçalho da função), e nós também temos que mudar o lugar de onde a função é chamada em `imprimeTabMult`.

Como esperado, este programa gera uma tabela quadrada de sete por sete:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

Quando você generaliza uma função apropriadamente, você muitas vezes tem um programa com capacidades que você não planejou. Por exemplo, você pode ter notado que, porque ab = ba, todas as entradas na tabela aparecem duas vezes. Você poderia economizar tinta imprimindo somente a metade da tabela. Para fazer isso, você tem que mudar apenas uma linha em `imprimeTabMult`. Mude:

```
imprimeTabMult(i, altura)
```

para:

```
imprimeTabMult(i, i)
```

e você terá:

```
1
2      4
3      6       9
4      8       12      16
5      10      15      20      25
6      12      18      24      30      36
7      14      21      28      35      42      49
```

Como um exercício, trace a execução desta versão de `imprimeTabMult` e explique como ela funciona.

8.9 6.9 Funções

- Há pouco tempo mencionamos "todas as coisas para as quais as funções são boas." Agora, você pode estar pensando que coisas exatamente são estas. Aqui estão algumas delas:
- Dar um nome para uma seqüência de comandos torna seu programa mais fácil de ler e de depurar.
- Dividir um programa longo em funções permite que você separe partes do programa, depure-as isoladamente, e então as componha em um todo.
- Funções facilitam tanto recursão quanto iteração.
- Funções bem projetadas são freqüentemente úteis para muitos programas. Uma vez que você escreva e depure uma, você pode reutilizá-la.

8.10 6.10 Glossário

reatribuição (*multiple assignment* [1]) quando mais de um valor é atribuído a mesma variável durante a execução do programa.

iteração (*iteration*) execução repetida de um conjunto de comandos/instruções (statements) usando uma chamada recursiva de função ou um laço (loop).

laço (*loop*) um comando/instrução ou conjunto de comandos/instruções que executam repetidamente até que uma condição de interrupção seja atingida.

laço infinito (*infinite loop*) um laço em que a condição de interrupção nunca será atingida.

corpo (*body*) o conjunto de comandos/instruções que pertencem a um laço.

variável de laço (*loop variable*) uma variável usada como parte da condição de interrupção do laço.

tabulação (*tab*) um carácter especial que faz com que o cursor mova-se para a próxima parada estabelecida de tabulação na linha atual.

nova-linha (*newline*) um carácter especial que faz com que o cursor mova-se para o início da próxima linha.

cursor (*cursor*) um marcador invisível que determina onde o próximo carácter var ser impresso.

sequência de escape (*escape sequence*) um carácter de escape () seguido por um ou mais caracteres imprimíveis, usados para definir um carácter não imprimível.

encapsular (*encapsulate*) quando um programa grande e complexo é dividido em componentes (como funções) e estes são isolados um do outro (pelo uso de variáveis locais, por exemplo).

generalizar (*generalize*) quando algo que é desnecessariamente específico (como um valor constante) é substituído por algo apropriadamente geral (como uma variável ou um parâmetro). Generalizações dão maior versatilidade ao código, maior possibilidade de reuso, e em algumas situações até mesmo maior facilidade para escrevê-lo.

plano de desenvolvimento (*development plan*) um processo definido para desenvolvimento de um programa. Neste capítulo, nós demonstramos um estilo de desenvolvimento baseado em escrever código para executar tarefas simples e específicas, usando encapsulamento e generalização.

Capítulo 7: Strings

Tópicos

9.1 7.1 Um tipo de dado composto

Até aqui, vimos três diferentes tipos de dado: `int`, `float` e `string`. Strings são qualitativamente diferentes dos outros dois tipos porque são feitas de pedaços menores - caracteres.

Tipos que consistem de pedaços menores são chamados **tipos de dados compostos**. Dependendo do que estejamos fazendo, pode ser que precisemos tratar um tipo de dado composto como uma coisa única, ou pode ser que queiramos acessar suas partes. Esta ambigüidade é útil.

O operador colchete seleciona um único caractere de uma string.:

```
>>> fruta = "banana"
>>> letra = fruta[1]
>>> print letra
```

A expressão `fruta[1]` seleciona o caractere número 1 de `fruta`. A variável `letra` referencia ou refere-se ao resultado da expressão. Quando exibimos `letra`, temos uma surpresa:

```
a
```

A primeira letra de `"banana"` não é `a`. A menos que você seja um cientista da computação. Neste caso, você deve entender a expressão dentro dos colchetes como um deslocamento (*offset*,) a partir do começo da string, e o deslocamento da primeira letra é zero. Assim, `b` é a 0ª ("zero-ésima") letra de `"banana"`, `a` é a 1ª ("um-ésima", diferente de primeira), e `n` é a 2ª ("dois-ésima", diferente de segunda) letra.

Para pegar a primeira letra de uma string, você simplesmente põe 0, ou qualquer expressão que resulte o valor 0, dentro dos colchetes:

```
>>> letra = fruta[0]
>>> print letra
b
```

A expressão entre colchetes é chamada de **índice**. Um índice especifica um membro de um conjunto ordenado, neste caso o conjunto de caracteres da string. O índice *indica* aquele membro que você quer, daí seu nome. Ele pode ser qualquer expressão inteira.

9.2 7.2 Comprimento

A função `len` retorna o número de caracteres de uma string:

```
>>> fruta = "banana"
>>> len(fruta)
6
```

Para pegar a última letra de uma string, você pode ficar tentado a fazer alguma coisa assim:

```
comprimento = len(fruta)
ultima = fruta[comprimento]    #  ERRO!
```

Não vai funcionar. Isto vai causar o seguinte erro em tempo de execução (*runtime error*): `IndexError: string index out of range`. (*ErroDeIndice: índice da string fora do intervalo*). A razão é que não existe 6ª letra em `"banana"`. Já que começamos a contar do zero, as seis letras são numeradas de 0 a 5. Para pegar o último caractere, temos que subtrair 1 de `comprimento`:

```
comprimento = len(fruta)
ultima = fruta[comprimento-1]
```

Como alternativa, podemos usar índices negativos, os quais contam de trás pra frente os elementos da string. A expressão `fruta[-1]` resulta a última letra, `fruta[-2]` resulta a penúltima (a segunda de trás para frente), e assim por diante.

9.3 7.3 Travessia e o loop `for`

Várias computações envolvem o processamento de uma string um caractere de cada vez. Muitas vezes elas começam com o primeiro, selecionam um de cada vez, fazem alguma coisa com ele, e continuam até o fim. Este padrão de processamento é chamado uma **travessia** (*traversal*, com a idéia de "percorrimento"). Uma maneira de codificar uma travessia é com um comando `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print letra
    indice = indice + 1
```

Este loop percorre a string e exibe cada letra em sua própria linha. A condição do loop é `indice < len(fruta)`, assim, quando `indice` é igual ao comprimento da string, a condição se torna falsa, e o corpo do loop não é executado. O último caractere acessado é aquele com o índice `len(fruta)-1`, que vem a ser o último caractere da string.

> Como um exercício, escreva uma função que tome uma string como argumento e devolva suas letras de trás para frente, uma por linha.

Usar um índice para percorrer um conjunto de valores é tão comum que Python oferece uma sintaxe alternativa simplificada - o loop `for`:

```
for char in fruta:
    print char
```

A cada vez através do loop, o próximo caractere da string é atribuído à variável `char`. O loop continua até que não reste mais caracteres.

O exemplo seguinte mostra como usar concatenação e um loop `for` para gerar uma série abecedário. "Abecedário" se refere a uma série ou lista na qual os elementos aparecem em ordem alfabética. Por exemplo, no livro de Robert McCloskey's *Make Way for Ducklings*, os nomes dos "*ducklings*" são Jack, Kack, Lack, Mack, Nack, Ouack, Pack e Quack. O loop seguinte, produz como saída aqueles nomes, em ordem:

```
prefixos = "JKLMNOPQ"
sufixo = "ack"

for letra in prefixos:
    print letra + sufixo
```

A saída deste programa é:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Naturalmente, esta saída não está cem por cento certa porque "Ouack" e "Quack" estão escritos de maneira errada.

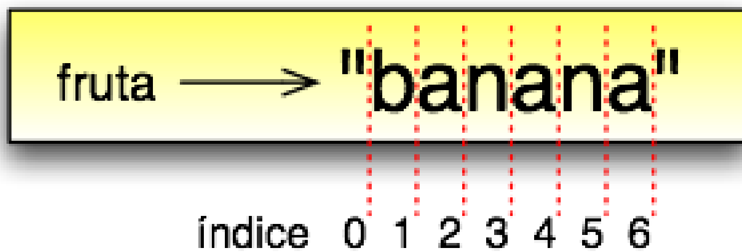
> Como um exercício, modifique o programa para corrigir este erro.

9.4 7.4 Fatias de strings

Um segmento de uma string é chamado de uma fatia. Selecionar uma fatia é similar a selecionar um caractere:

```
>>> s = "Pedro, Paulo e Maria"
>>> print s[0:5]
Pedro
>>> print s[7:12]
Paulo
>>> print s[16:21]
Maria
```

O operador `[n:m]` retorna a parte da string do "n-ésimo" caractere ao "m-ésimo" caractere, incluindo o primeiro mas excluindo o último. Este comportamento não é intuitivo; ele faz mais sentido se você imaginar os índices apontando para os intervalos *entre* os caracteres, como no seguinte diagrama:



Se você omitir o primeiro índice (antes dos dois pontos ":"), a fatia começa do início da string. Se você omitir o segundo índice, a fatia vai até o final da string. Assim:

```
>>> fruta = "banana"
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

O que você acha de `s[:]` significa?

9.5 7.5 Comparação de strings

O operador de comparação funciona com strings. Para ver se duas strings são iguais:

```
if palavra == "banana":
    print "Sim, nós não temos bananas!"
```

Outras operações de comparação são úteis para colocar palavras em ordem alfabética:

```
if palavra < "banana":
    print "Sua palavra," + palavra + ", vem antes de banana."
elif palavra > "banana":
    print "Sua palavra," + palavra + ", vem depois de banana."
else:
    print "Sim, nós não temos bananas!"
```

Entretanto, você deve atentar para o fato de que Pyhton não manipula letras maiúsculas e minúsculas da mesma maneira que as pessoas o fazem. Todas as letras maiúsculas vêm antes das minúsculas. Como resultado:

```
Sua palavra, Zebra, vem antes de banana.
```

Uma maneira comum de resolver este problema é converter as strings para um formato padrão, seja todas minúsculas, ou todas maiúsculas, antes de realizar a comparação. Um problema mais difícil é fazer o programa perceber que zebras não são frutas.

9.6 7.6 Strings são imutáveis

É tentador usar o operador `[]` no lado esquerdo de uma expressão de atribuição, com a intenção de alterar um caractere em uma string. Por exemplo:

```
saudacao = "Alô, mundo!"
saudacao[0] = 'E'            #  ERRO!
print saudacao
```

Em vez de produzir a saída `Elô, Mundo!`, este código produz o erro em tempo de execução (*runtime error*): `TypeError: object doesn't support item assignment` (*ErroDeTipo: objeto não dá suporte à atribuição de item.*)

Strings são **imutáveis**, o que significa que você não pode mudar uma string que já existe. O melhor que você pode fazer é criar uma nova string que seja uma variação da original:

```
saudacao = "Alô, mundo!"
novaSaudacao = 'E' + saudacao[1:]
print novaSaudacao
```

A solução aqui é concatenar uma nova primeira letra com uma fatia de `saudacao`. Esta operação não tem nenhum efeito sobre a string original.

9.7 7.7 Uma função `find` (*encontrar*)

O que faz a seguinte função?:

```
def find(str, ch):
    indice = 0
    while indice < len(str):
        if str[indice] == ch:
            return indice
        indice = indice + 1
    return -1
```

Num certo sentido, `find` (*encontrar*) é o oposto do operador `[]`. Em vez de pegar um índice e extrair o caractere correspondente, ela pega um caractere e encontra (*finds*) em qual índice aquele caractere aparece. Se o caractere não é encontrado, a função retorna `-1`.

Este é o primeiro exemplo que vemos de uma instrução `return` dentro de um loop. Se `str[indice] == ch`, a função retorna imediatamente, abandonando o loop prematuramente.

Se o caractere não aparece na string, então o programa sai do loop normalmente e retorna `-1`.

Este padrão de computação é às vezes chamado de travessia "eureka", porque tão logo ele encontra (*find*) o que está procurando, ele pode gritar "Eureka!" e parar de procurar.

> Como um exercício, modifique a função find (encontrar) de modo que ela receba um terceiro parâmetro, o índice da string por onde ela deve começar sua procura.

9.8 7.8 Iterando e contando

O programa seguinte conta o número e vezes que a letra a aparece em uma string:

```
fruta = "banana"
contador = 0
for letra in fruta:
  if letra == 'a':
    contador = contador + 1
print contador
```

Este programa demonstra um outro padrão de computação chamado de **contador**. A variável `contador` é inicializada em `0` e então incrementada cada vez que um `a` é encontrado. (**Incrementar** é o mesmo que aumentar em um; é o oposto de **decrementar**, e não tem relação com excremento, que é um substantivo.) Quando se sai do loop, `contador` guarda o resultado - o número total de `a`'s.

Como um exercício, encapsule este código em uma função chamada `contaLetras`, e generalize-a de modo que possa aceitar uma string e uma letra como parâmetros.

Como um segundo exercício, reescreva esta função de modo que em vez de percorrer a string, ela use a versão com três parâmetros de `find` (encontrar) da seção anterior.

9.9 7.9 O módulo `string`

O módulo `string` contém funções úteis que manipulam strings. Conforme é usual, nós temos que importar o módulo antes que possamos utilizá-lo:

```
>>> import string
```

O módulo `string` inclui uma função chamada `find` (encontrar) que faz a mesma coisa que a função que escrevemos. Para chamá-la, temos que especificar o nome do módulo e o nome da função usando a notação de ponto.:

```
>>> fruta = "banana"
>>> indice = string.find(fruta, "a")
>>> print indice
1
```

Este exemplo demonstra um dos benefícios dos módulos - eles ajudam a evitar colisões entre nomes de funções nativas e nomes de funções definidas pelo usuário. Usando a notação de ponto podemos especificar que versão de `find` (*encontrar*) nós queremos.

De fato, `string.find` é mais generalizada que a nossa versão. Primeiramente, ela pode encontrar substrings, não apenas caracteres:

```
>>> string.find("banana", "na")
2
```

Além disso, ela recebe um argumento adicional que especifica o índice pelo qual ela deve começar sua procura:

```
>>> string.find("banana", "na", 3)
4
```

Ou ela pode receber dois argumentos adicionais que especificam o intervalo de índices:

```
>>> string.find("bob", "b", 1, 2)
-1
```

Neste exemplo, a busca falha porque a letra b não aparece no intervalo entre 1 e 2 (não incluindo o 2) do índice.

9.10 7.10 Classificação de caracteres

Muitas vezes é útil examinar um caractere e testar se ele é maiúsculo ou minúsculo, ou se ele é um caractere ou um dígito. O módulo `string` oferece várias constantes que são úteis para esses propósitos.

A string `string.lowercase` contém todas as letras que o sistema considera como sendo minúsculas. Similarmente, `string.uppercase` contém todas as letras maiúsculas. Tente o seguinte e veja o que você obtém:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

Nós podemos usar essas constantes e `find` (encontrar) para classificar caracteres. Por exemplo, se `find(lowercase, ch)` retorna um valor outro que não `-1`, então `ch` deve ser minúsculo:

```
def eMinusculo(ch):
    return string.find(string.lowercase, ch) != -1
```

Como uma alternativa, podemos tirar vantagem do operador `in`, que determina se um caractere aparece em uma string:

```
def eMinusculo(ch):
    return ch in string.lowercase
```

Ainda, como uma outra alternativa, podemos usar o operador de comparação:

```
def eMinusculo(ch):
    return 'a' <= ch <= 'z'
```

Se `ch` estiver entre *a* e *z*, ele deve ser uma letra minúscula.

> Como um exercício, discuta que versão de `eMinusculo` você acha que será a mais rápida. Você pode pensar em outras razões além da velocidade para preferir uma em vez de outra?

Outra constante definida no módulo `string` pode te surpreender quando você executar um `print` sobre ela:

```
>>> print string.whitespace
```

Caracteres de **espaçamento** (ou *espaços em branco*) movem o cursor sem "imprimir" qualquer coisa. Eles criam os espaços em branco entre os caracteres visíveis (pelo menos numa folha de papel branco). A string constante `string.whitespace` contém todos os caracteres de espaçamento, incluindo espaço, tabulação (`\t`) e nova linha (`\n`).

Existem outras funções úteis no módulo `string`, mas este livro não pretende ser um manual de referência. Por outro lado, *Python Library Reference* é exatamente isto. Em meio a uma abundante documentação, ele está disponível no site da web do Python, `www.python.org`.

9.11 7.11 Glossário

tipo de dado composto (*compound data type*) Um tipo de dado no qual o valor consiste de componentes, ou elementos, que são eles mesmos valores.

travessia (*traverse*) Iterar através dos elementos de um conjunto, realizando uma operação similar em cada um deles.

índice (*index*) Uma variável ou valor usados para selecionar um membro de um conjunto ordenado, como um caractere em uma string.

fatia (*slice*) Uma parte de uma string especificada por um intervalo de índices.

mutável (*mutable*) Um tipo de dado composto a cujos elementos podem ser atribuídos novos valores.

contador (*counter*) Uma variável utilizada para contar alguma coisa, usualmente inicializada em zero e então incrementada.

incrementar (*increment*) aumentar o valor de uma variável em 1.

decrementar (*decrement*) diminuir o valor de uma variável em 1.

espaçamento (*whitespace*) Qualquer um dos caracteres que move o cursor sem imprimir caracteres visíveis. A constante `string.whitespace` contém todos os caracteres de espaçamento.

9.12 7.11 Glossário2

tipo de dado composto (*compound data type*) Um tipo de dado em que os valores são compostos de componentes, ou elementos, que podem ser tratados como valores separados.

atravessar (*traverse*) Iterar através dos elementos definidos, executando uma operação similar em cada.

índice (*index*) Uma variável ou valor usado para selecionar um membro de uma definição ordenada, como um caractere de uma string.

fatia (*slice*) Uma parte da string especificada por um intervalo de índice.

mutável (*mutable*) Um tipo de dado composto do qual elementos podem atribuir novos valores.

contador (*counter*) Uma variável usada para contar alguma coisa, geralmente iniciada em zero e incrementada.

incremento (*increment*) Para aumentar o valor da variável.

decremento (*decrement*) Para dimiuir o valor da variável.

espaço em branco (*whitespace*) Qualquer caractere que move o cursor sem imprimir caracteres visíveis. A constante string.whitespace contém todos os caracteres de espaço em branco.

Capítulo 8: Listas

Tópicos

Uma **lista** é um conjunto ordenado de valores, onde cada valor é identificado por um índice. Os valores que compõem uma lista são chamados **elementos**. Listas são similares a strings, que são conjuntos ordenados de caracteres, com a diferença que os elementos de uma lista podem possuir qualquer tipo. Listas e strings XXX e outras coisas que se comportam como conjuntos ordenados XXX são chamados **seqüências**.

10.1 8.1 Valores da lista

Existem várias maneiras de criar uma nova lista; a mais simples é envolver os elementos em colchetes (`[` e `]`):

```
>>> [10, 20, 30, 40]
>>> ['spam', 'bungee', 'swallow']
```

O primeiro exemplo á uma lista de quatro inteiros. O segundo é uma lista de três strings. Os elementos de uma lista não necessitam ser do mesmo tipo. A lista a seguir contém uma string, um valor *float*, um valor inteiro, e *mirabile dictu* uma outra lista:

```
>>> ['alo', 2.0, 5, [10,20]]
```

Uma lista dentro de outra lista é dita estar **aninhada**.

Listas que contém inteiros consecutivos são comuns, então Python fornece uma maneira simples de criá-los:

```
>>> range(1,5)
[1, 2, 3, 4]
```

A função `range` pega dois argumentos e devolve uma lista que contém todos os inteiros do primeiro até o segundo, incluindo o primeiro mas não incluindo o segundo!

Existem outras formas de `range`. Com um argumento simples, ela cria uma lista que inicia em 0:

```
>>> range(10)
[0,1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se existe um terceiro argumento, ele especifica o espaço entre os valores sucessivos, que é chamado de `tamanho do passo`. Este exemplo conta de 1 até 10 em passos de 2:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Finalmente, existe uma lista especial que não contém elementos. Ela é chamada lista vazia, e sua notação é `[]`.

Com todas estas formas de criar listas, seria decepcionante se não pudéssemos atribuir valores de listas a variáveis ou passar listas como parâmetros a funções. Felizmente, podemos.

```
>>> vocabulario = ['melhorar', 'castigar', 'defenestrar']
>>> numeros = [17, 123]
>>> vazio = []
>>> print vocabulario, numeros, vazio
['melhorar', 'castigar', 'defenestrar'] [17, 123] []
```

10.2 8.2 Acessado elementos

A sintaxe para acessar os elementos de uma lista é a mesma que a sintaxe para acessar os caracteres de uma string XXX o operator colchete (`[]`). A expressão dentro dos colchetes especifica o índice. Lembre-se que os índices iniciam em 0:

```
>>> print numeros[0]
>>> numeros[1]= 5
```

O operador colchete pode aparecer em qualquer lugar em uma expressão. Quando ele aparece no lado esquerdo de uma atribuição, ele modifica um dos elementos em uma lista, de forma que o um-ésimo elemento de `numeros`, que era `123`, é agora `5`.

Qualquer expressão inteira pode ser utilizada como um índice:

```
>>> numeros[3-2]
5
>>> numeros[1.0]
TypeError: sequence index must be integer
```

Se você tentar ler ou escrever um elemento que não existe, você recebe um erro de tempo de execução (*runtime error*):

```
>>> numeros[2]=5
IndexError: list assignment index out of range
```

Se um índice possui um valor negativo, ele conta ao contrário a partir do final da lista:

```
>>> numeros[-1]
5
>>> numeros[-2]
17
>>> numeros[-3]
IndexError: list index out of range
```

`numeros[-1]` é o último elemento da lista, `numeros[-2]` é o penúltimo e `numeros[-3]` não existe.

É comum utilizar uma variável de laço como um índice da lista:

```
>>> cavaleiros = ['guerra', 'fome', 'peste', 'morte']
i = 0
while i < 4:
        print cavaleiros[i]
        i = i + 1
```

Este laço `while` conta de 0 até 4. Quando a variável do laço `i` é 4, a condição falha e o laço se encerra. Desta forma o corpo do laço é executado somente quando `i` é 0, 1, 2 e 3.

Em cada vez dentro do laço, a variável `i` é utilizada como um índice para a lista, exibindo o `i-ésimo` elemento. Este padrão de computação é chamado de **percurso na lista**.

10.3 8.3 Comprimento da lista

A função `len` devolve o comprimento de uma lista. É uma boa idéia utilizar este valor como o limite superior de um laço ao invés de uma constante. Desta forma, se o tamanho da lista mudar, você não precisará ir através de todo o programa modificando todos os laços; eles funcionarão corretamente para qualquer tamanho de lista:

```
>>> cavaleiros = ['guerra', 'fome', 'peste', 'morte']
i = 0
while i < len(cavaleiros):
        print cavaleiros[i]
        i = i + 1
```

A última vez que o corpo do laço é executado, `i` é `len(cavaleiros) - 1`, que é o índice do último elemento. Quando `i` é igual a `len(cavaleiros)`, a condição falha e o corpo não é executado, o que é uma boa coisa, porque `len(cavaleiros)` não é um índice legal.

Embora uma lista possa conter uma outra lista, a lista aninhada ainda conta como um elemento simples. O comprimento desta lista é quatro:

```
>>> [`spam!', 1, ['Brie', 'Roquefort', 'Pol lê Veq'], [1, 2 3]]
```

> Como um exercício, escreva um laço que percorra a lista anterior e exiba o comprimento de cada elemento. O que acontece se você manda um inteiro para `len`?

10.4 8.4 Membros de uma lista

`in` é um operador lógico que testa se um elemento é membro de uma seqüência. Nós o utilizamos na `Seção 7.10` com strings, mas ele também funciona com listas e outras seqüências:

```
>>> cavaleiros = ['guerra', 'fome', 'peste', 'morte']
>>> 'peste' in cavaleiros
True
>>> 'depravação' in cavaleiros
False
```

Uma vez que 'peste' é um membro da lista `cavaleiros`, o operador `in` devolve `verdadeiro`. Uma vez que `depravação` não está na lista, `in` devolve `falso`.

Podemos utilizar também o `not` em combinação com o `in` para testar se um elemento não é um membro de uma lista:

```
>>> ``depravação`` not in cavaleiros
True
```

10.5 8.5 Listas e laços `for`

O laço `for` que vimos na Seção 7.3 também funciona com listas. A sintaxe generalizada de um laço `for` é:

```
for VARIAVEL in LISTA:
        CORPO
```

Esta declaração é equivalente a:

```
>>> i = 0
>>> while i < len(LISTA):
>>>       VARIAVEL = LISTA[i]
>>>       XXX BODY
>>>   i = i + 1
```

O laço `for` é mais conciso porque podemos eliminar a variável do laço, `i`. Aqui está o laço anterior escrito com um'laço `for`:

```
>>> for cavaleiro in cavaleiros:
        print cavaleiro
```

Quase se lê como Português: "For (para cada) cavaleiro in (na lista de) cavaleiros, print (imprima o nome do) cavaleiro."

Qualquer expressão de lista pode ser utilizada num laço `for`:

```
>>> for numero in range(20):
        if numero % 2 == 0:
                print numero

>>> for fruta in ["banana", "abacaxi", "laranja"]:
        print "Eu gosto de comer " + fruta + "s!"
```

O primeiro exemplo exibe todos os números pares entre zero e dezenove. O segundo exemplo expressa o entusiasmo por várias frutas.

10.6 8.6 Operações em listas

O operador + concatena listas:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarmente, o operador * repete uma lista um número dado de vezes:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

O primeiro exemplo repete `[0]` quatro vezes. O segundo exemplo repete a lista `[1, 2, 3]` três vezes.

10.7 8.7 Fatiamento de listas

A operação de fatiamento que vimos na Seção 7.4 também funciona sobre listas:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3]
['b', 'c']
>>> lista[:4]
['a', 'b', 'c', 'd']
>>> lista[3:]
['d', 'e', 'f']
>>> lista[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

10.8 8.8 Listas são mutáveis

Diferente das strings, as listas são mutáveis, o que significa que podemos modificar seus elementos. Utilizando o operador colchete no lado esquerdo de uma atribuição, podemos atualizar um de seus elementos:

```
>>> fruta = ["banana", "abacaxi", "laranja"]
>>> fruta[0] = "abacate"
>>> fruta[-1] = "tangerina"
>>> print fruta
['abacate', 'abacaxi', 'tangerina']
```

Com o operador de fatiamento podemos atualizar vários elementos de uma vez:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = ['x', 'y']
>>> print lista
['a', 'x', 'y', 'd', 'e', 'f']
```

Também podemos remover elementos de uma lista atribuindo a lista vazia a eles:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = []
>>> print lista
['a', 'd', 'e', 'f']
```

E podemos adicionar elementos a uma lista enfiando-os numa fatia vazia na posição desejada:

```
>>> lista = ['a', 'd', 'f']
>>> lista[1:1] = ['b', 'c']
>>> print lista
['a', 'b', 'c', 'd', 'f']
>>> lista[4:4] = ['e']
>>> print lista
['a', 'b', 'c', 'd', 'e', 'f']
```

10.9 8.9 Remoção em lista

Utilizando fatias para remover elementos pode ser complicado, e desta forma propenso a erro. Python fornece uma alternativa que é mais legível.

`del` remove um elemento de uma lista:

```
>>> a = ['um', 'dois', 'tres']
>>> del a[1]
>>> a
['um', 'tres']
```

Como você deveria esperar, `del` trata valores negativos e causa erros de tempo de execução se o índice estiver fora da faixa.

Você também pode utilizar uma faixa como um índice para `del`:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del lista[1:5]
>>> print lista
['a', 'f']
```

Como de costume, fatias selecionam todos os elementos até, mas não incluindo, o segundo índice.

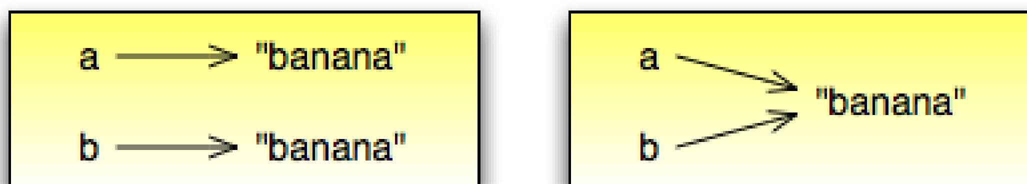
10.10 8.10 Ojetos e valores

Se executamos estas declarações de atribuição:

```
>>> a = "banana"
>>> b = "banana"
```

sabemos que `a` e `b` se referem a uma string com as letras `banana`. Mas não podemos dizer se elas apontam para *a mesma* string.

Existem dois possíveis estados:



Em um caso, `a` e `b` se referem a duas coisas diferentes que possuem o mesmo valor. No segundo caso, elas se referem à mesma coisa. Estas "coisas" possume nomes - elas são chamadas **objetos**. Um objeto é algo ao qual uma variável pode se referenciar.

Todo objeto possui um **identificador** único, que podemos obter com a função `id`. Exibindo o identificador de `a` e `b`, podemos dizer se elas se referem ao mesmo objeto.

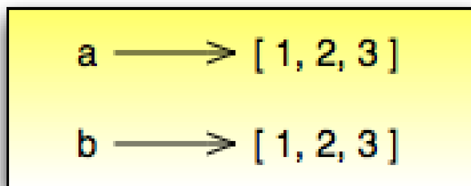
```
>>> id(a)
135044008
>>> id(b)
135044008
```

De fato, obtivemos o mesmo identificador duas vezes, o que significa que Python criou apenas uma string, e tanto `a` quanto `b` se referem a ela.

Interessantemente, listas se comportam de forma diferente. Quando criamos duas listas, obtemos dois objetos:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

Então o diagrama de estado fica assim:



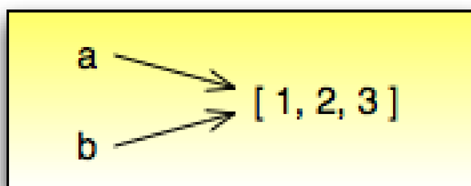
`a` e `b` possuem o mesmo valor mas não se referem ao mesmo objeto.

10.11 8.11 Apelidos

Uma vez que variáveis se referem a objetos, se atribuimos uma variável a uma outra, ambas as variáveis se referem ao mesmo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
```

Neste caso, o diagrama de estado se parece com isto:



Uma vez que a lista possui dois nomes diferentes, `a` e `b`, dizemos que ela está "apelidada" (aliased). Mudanças feitas em um apelido afetam o outro nome:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Embora este comportamento possa ser útil, ele é às vezes inesperado e indesejado. Em geral, é mais seguro evitar os apelidos quando você está trabalhando com objetos mutáveis. É claro, para objetos imutáveis, não há problema. É por isto que Python é livre para apelidar cadeias de caracteres quando vê uma oportunidade de economizar.

10.12 8.12 Clonando listas

Se queremos modificar uma lista e também manter uma cópia da original, preciamos ter condições de fazer uma cópia da própria lista, não apenas uma referência. Este processo é algumas vezes chamado **clonagem**, para evitar a ambigüidade da palavra "cópia".

A maneira mas fácil de clonar uma lista é utilizar o operador de fatia:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Pegar qualquer fatia de `a` cria uma nova lista. Neste caso acontece da fatia consistir da lista inteira.

Agora estamos livres para fazer alterações a `b` sem nos preocuparmos com‘‘a‘‘:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

> Como exercício, desenhe um diagrama de estado para‘‘a‘‘ e `b` antes e depois desta mudança.

10.13 8.13 Lista como parâmetro

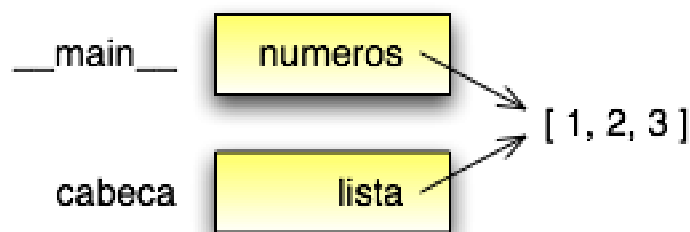
Passar uma lista como um argumento passa realmente uma referência à lista, não uma cópia da lista. Por exemplo, a função `cabeca` pega uma lista como parâmetro e devolve a cabeça da lista, ou seja, seu primeiro elemento:

```
>>> def cabeca(lista):
        return lista[0]
```

Eis como ela é utilizada:

```
>>> numeros = [1, 2, 3]
>>> cabeca(numeros)
1
```

O parâmetro `lista` e a variável `numeros` são apelidos para o mesmo objeto. O diagrama de estado se parece com isto:



Uma vez que o objeto é compartilhado pelos dois quadros, o desenhamos entre eles.

Se a função modifica um parâmetro da lista, a função chamadora vê a mudança. Por exemplo, `removeCabeca` remove o primeiro elemento da lista:

```
>>> def removecabeca(lista):
        del lista[0]
```

Aqui está a maneira como ela é utilizada:

```
>>> numeros = [1, 2, 3]
>>> removeCabeca(numeros)
>>> print numeros
[2, 3]
```

Se uma função devolve uma lista, ela devolve uma referência à lista. Por exemplo, `cauda` devolve uma lista que contém todos menos o primeiro elemento de uma determinada lista:

```
>>> def cauda(lista):
        return lista[1:]
```

Aqui está a maneira como ela é utilizada:

```
>>> numeros = [1, 2, 3]
>>> resto = cauda(numeros)
>>> print resto
[2, 3]
```

Uma vez que o valor de retorno foi criado com o operador de fatia, ele é uma nova lista. A criação de `resto`, e qualquer alteração subseqüente a `resto`, não tem efeito sobre `numeros`.

10.14 8.14 Lista aninhadas

Uma lista aninhada é uma lista que aparece como um elemento de uma outra lista. Nesta lista, o terceiro elemento é uma lista aninhada:

```
>>> lista = ["alo", 2.0, 5, [10, 20]]
```

Se exibimos `lista[3]`, obtemos `[10, 20]`. Para extrairmos um elemento de uma lista aninhada, podemos agir em duas etapas:

```
>>> elem = lista[3]
>>> elem[0]
10
```

Ou podemos combiná-las:

```
>>> lista[3][1]
20
```

Os operadores colchete avaliam da esquerda para a direita, então a expressão pega o terceiro elemento de `lista` e extrai o primeiro elemento dela.

10.15 8.15 Matrizes

Listas aninhadas são freqüentemente utilizadas para representar matrizes. Por exemplo, a matriz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

poderia ser representada como:

```
>>> matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`matriz` é uma lista com três elementos, onde cada elemento é uma linha da matriz. Podemos selecionar uma linha inteira da matriz da maneira habitual:

```
>>> matriz[1]
[4, 5, 6]
```

Ou podemos extrair um único elemento da matriz utilinado a forma de duplo índice:

```
>>> matriz[1][1]
5
```

O primeiro índice seleciona a linha, e o segundo índice seleciona a coluna. Embora esta maneira de representar matrizes seja comum, ela não é a única possibilidade. Uma pequena variação é utilizar uma lista de colunas ao invés de uma lista de linhas.

Mais adiante veremos uma alternativa mais radical utilizando um dicionário.

10.16 8.16 Strings e listas

Duas das mais úteis funções no módulo `string` envolvem listas de strings. A função `split` (separar) quebra uma string em uma lista de palavras. Por padrão, qualquer número de caracteres espaço em branco é considerado um limite de uma palavra:

```
>>> import string
>>> poesia = "O orvalho no carvalho..."
>>> string.split(poesia)
['O', 'orvalho', 'no', 'carvalho...']
```

Um argumento opcional chamado um **delimitador** pode ser utilizado para especificar qual caracter utilizar como limites da palavra. O exemplo a seguir utiliza a string `va`:

```
>>> string.split(poesia, 'va')
['O or', 'lho no car', 'lho...']
```

Perceba que o delimitador não aparece na lista.

A função `join` (juntar) é o inverso de `split`. Ela pega uma lista de strings e concatena os elementos com um espaço entre cada par:

```
>>> lista = ['O', 'orvalho', 'no', 'carvalho...']
>>> string.join(lista)
'O orvalho no carvalho...'
```

Como `split`, `join` recebe um delimitador que é inserido entre os elementos:

```
>>> string.join(lista, '_')
'O_orvalho_no_carvalho...'
```

Como um execício, descreva o relacionamento entre `string.join(string.split(poesia))` e `poesia`. Eles são o mesmo para qualquer string? Quando eles seriam diferentes?

10.17 8.17 Glossário

lista (*list*) Uma coleção *denominada* de objetos, onde cada objeto é identificado por um índice.

índice (*index*) Uma variável inteira ou valor que indica um elemento de uma lista.

elemento (*element*) Um dos valores em uma lista(ou outra seqüência). O operador colchete seleciona elementos de uma lista.

seqüência (*sequence*) Qualquer um dos tipos de dados que consiste de um conjunto ordenado de elementos, com cada elemento identificado por um índice.

lista aninhada (*nested list*) Uma lista que é um elemento de uma outra lista.

percurso na lista (*list traversal*) O acesso seqüencial de cada elemento em uma lista.

objeto (*object*) Um coisa a qual uma variável pode se referir.

apelidos (*aliases*) Múltiplas variáveis que contém referências ao mesmo objeto.

clonar (*clone*) Criar um novo objeto que possui o mesmo valor de um objeto existente. Copiar a referência a um objeto cria um apelido (*alias*) mas não clona o objeto.

delimitador (*delimiter*) Um caracter uma string utilizados para indicar onde uma string deveria ser dividida(*split*).

10.18 Outros termos utilizados neste capítulo

(XXX esta lista deve ser retirada na versão final)

XXX *has, have* possuir (ter?)

XXX *there is, there are* existir (haver?)

XXX *use* utilizar (usar?)

XXX string Utilizei string em itálico, por ser tratar de um termo que não é em português.

XXX *enclose* envolver???

XXX *provide* fornecer

XXX *return* devolve

XXX *denoted* denotada XXX

XXX *disappointing* decepcionante (desapontador?)

XXX *assign* atribuir

XXX *change* modificar

XXX *length* comprimento (tamanho?)

XXX *print* exibir (imprimir?)

XXX *membership* Não creio que exista uma palavra que traduza este termo. Pelo menos em inglês não encontrei nenhum sinônimo. Vou tentar traduzir explicando o termo dependendo do contexto.

XXX *boolean* lógico (booleano?)

XXX handle tratar

XXX proceed agir

XXX By default por padrão

XXX notice perceber (observar?)

XXX *mirabile dictu* Alguém tem idéia do que significa isto? Meu latim não chegou lá. :)

XXX *traduzir os exemplos?* considero melhor fazer a traduzir os exemplos sempre que possível. Só não gostaria de tirar o espírito que levou o autor a utilizar tais exemplos. Podem haver trocadilhos, homenagens e outros sentimentos no autor que não devemos retirar. Desta forma, estou traduzindo todos os termos que consigo entender e encontrar palavras que exprimam a idéia. Nos demais, estou mantendo os termos originais para uma discussão futura.

CAPÍTULO 11

Capítulo 9: Tuplas

Tópicos

11.1 9.1 Mutabilidade e tuplas

Até agora, você tem visto dois tipos compostos: strings, que são compostos de caracteres; e listas, que são compostas de elementos de qualquer tipo. Uma das diferenças que notamos é que os elementos de uma lista podem ser modificados, mas os caracteres em uma string não. Em outras palavras, strings são **imutáveis** e listas são **mutáveis**.

Há um outro tipo em Python chamado **tupla** *(tuple)* que é similar a uma lista exceto por ele ser **imutável**. Sintaticamente, uma tupla é uma lista de valores separados por vírgulas:

```
>>> tupla = 'a', 'b', 'c', 'd', 'e'
```

Embora não seja necessário, é convencional colocar tuplas entre parênteses:

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
```

Para criar uma tupla com um único elemento, temos que incluir uma vírgula final:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Sem a vírgula, Python entende `('a')` como uma string entre parênteses:

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

Questões de sintaxe de lado, as operações em tuplas são as mesmas operações das listas. O operador índice seleciona um elemento da tupla.

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
>>> tupla[0]
'a'
```

E o operador *slice* (fatia) seleciona uma "faixa" (*range*) de elementos.

```
>>> tupla[1:3]
('b', 'c')
```

Mas se tentarmos modificar um dos elementos de uma tupla, teremos um erro:

```
>>> tupla[0] = 'A'
TypeError: object doesn't support item assignment
```

Naturalmente, mesmo que não possamos modificar os elementos de uma tupla, podemos substituí-la por uma tupla diferente:

```
>>> tupla = ('A',) + tupla[1:]
>>> tupla
('A', 'b', 'c', 'd', 'e')
```

11.2 9.2 Atribuições de tupla

De vez em quando, é necessário trocar entre si os valores de duas variáveis. Com operações de atribuição convencionais, temos que utilizar uma variável temporária. Por exemplo, para fazer a troca entre `a` e `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Se você tiver que fazer isso com frequência, esta abordagem se torna incômoda. Python fornece uma forma de **atribuição de tupla** que resolve esse problema elegantemente:

```
>>> a, b = b, a
```

O lado esquedo é uma tupla de variáveis; o lado direito é uma tupla de valores. Cada valor é atribuído à sua respectiva variável. Todas as expressões do lado direito são avaliadas antes de qualquer das atribuições. Esta característica torna as atribuições de tupla bastante versáteis.

Naturalmente, o número de variáveis na esquerda e o número de valores na direita deve ser igual:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

11.3 9.3 Tuplas como valores de retorno

Funções podem retornar tuplas como valor de retorno. Por Exemplo, nós poderíamos escrever uma função que troca dois parâmetros entre si:

```
def troca(x, y):
  return y, x
```

Então nós poderíamos atribuir o valor de retorno para uma tupla com duas variáveis:

```
a, b = troca(a, b)
```

Neste caso, não existe uma grande vantagem em fazer de `troca` (*swap*) uma função. De fato, existe um perigo em tentar encapsular `troca`, o qual é a tentação de cometer o seguinte erro:

```
def troca(x, y):      # versao incorreta
  x, y = y, x
```

Se nós chamarmos esta função desta forma:

```
troca(a, b)
```

então `a` e `x` são apelidos para um mesmo valor. Mudar `x` dentro da função `troca`, faz com que `x` se referencie a um valor diferente, mas sem efeito sobre `a` dentro de `__main__`. Do mesmo modo, a mudança em `y` não tem efeito sobre `b`.

Esta função roda sem produzir uma mensagem de erro, mas ela não faz o que pretendemos. Este é um exemplo de um erro semântico.

> *Como exercício, desenhe um diagrama de estado pra esta função de modo que você possa ver porque ela não funciona.*

11.4 9.4 Números aleatórios

A maioria dos programas de computador fazem a mesma coisa sempre que são executados, então, podemos dizer que eles são **determinísticos**. Determinismo em geral é uma coisa boa, se nós esperamos que um cálculo dê sempre o mesmo resultado. Entretanto, para algumas aplicações queremos que o computador se torne imprevisível. Jogos são um exemplo óbvio, mas existem outros.

Fazer um programa realmente não-determinístico se mostra não ser tão fácil, mas existem maneiras de fazê-lo ao menos parecer não-determinístico. Uma dessas maneiras é gerar números aleatórios e usá-los para determinar o resultado de um programa. Python tem uma função nativa que gera números **pseudo aleatórios**, os quais não são verdadeiramente aleatórios no sentido matemático, mas para os nossos propósitos eles são.

O módulo `random` contém uma função chamada `random` que retorna um número em ponto flutuante (*floating-point number*) entre 0.0 e 1.0. Cada vez que você chama `random`, você recebe o próximo número de uma longa série. Para ver uma amostra, execute este loop:

```
import random

for i in range(10):
  x = random.random()
  print x
```

Para gerar um número aleatório ente 0.0 e um limite superior, digamos `superior`, multiplique `x` por `superior`.

> *Como exercício, gere um número aleatório entre 'inferior' e 'superior'.*
>
> *Como exercício adicional, gere um número inteiro aleatório entre 'inferior' e 'superior', inclusive os dois extremos.*

11.5 9.5 Lista de números aleatórios

O primeiro passo é gerar uma lista aleatória de valores. `listaAleatoria` pega um parâmetro inteiro e retorna uma lista de números aleatórios com o comprimento dado. Inicia-se com uma lista de `n` zeros. A cada iteração do loop, ele substitui um dos elementos por um número aleatório. O valor retornado é uma referência para a lista completa:

```
def listaAleatoria(n):
  s = [0] * n
  for i in range(n):
    s[i] = random.random()
  return s
```

Vamos realizar um teste desta função com uma lista de oito elementos. Para efeitos de depuração, é uma boa idéia começar com uma lista pequena.

```
>>> listaAleatoria(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

Os números gerados por `random` são supostamente uniformemente distribuídos, o que significa que cada valor tem uma probabilidade igual de acontecer.

Se nós dividirmos a faixa de valores possíveis em intervalos do mesmo tamanho, e contarmos o número de vezes que um determinado valor aleatório caiu em seu respectivo intervalo, nós devemos obter o mesmo número aproximado de valores em cada um dos intervalos.

Nós podemos testar esta teoria escrevendo um programa que divida a faixa de valores em intervalos e conte o número de valores de cada intervalo.

11.6 9.6 Contando

Uma boa maneira de abordar problemas como esse é dividir o problema em subproblemas, e encontrar um subproblema que se enquadre em um padrão de solução computacional que você já tenha visto antes.

Neste caso, queremos percorrer uma lista de números e contar o número de vezes que valor se encaixa em um determinado intervalo. Isso soa familiar. Na Seção 7.8, nós escrevemos um programa que percorria uma string e contava o número de vezes que uma determinada letra aparecia.

Assim, podemos prosseguir copiando o programa original e adaptando-o para o problema atual. O programa original era:

```
contador = 0
for letra in fruta:
  if letra == 'a':
    contador = contador + 1
print contador
```

O primeiro passo é substituir `fruta` por `lista` e `letra` por `numero`. Isso não muda o programa, apenas o ajusta para que ele se torne mais fácil de ler e entender.

O segundo passo é mudar o teste. Nós não estamos interessados em procurar letras. Nós queremos ver se `numero` está entre `inferior` e `superior`.:

```
contador = 0
  for numero in lista
    if inferior < numero < superior:
      contador = contador + 1
  print contador
```

O último passo é encapsular este código em uma função chamada `noIntervalo`. Os parâmetros são a lista e os valores `inferior` e `superior`:

```
def noIntervalo(lista, inferior, superior):
    contador = 0
    for numero in lista:
      if inferior < numero < superior:
        contador = contador + 1
    return contador
```

Através da cópia e da modificação de um programa existente, estamos aptos a escrever esta função rapidamente e economizar um bocado de tempo de depuração. Este plano de desenvolvimento é chamado de **casamento de padrões**. Se você se encontrar trabalhando em um problema que você já solucionou antes, reuse a solução.

11.7 9.7 Vários intervalos

Conforme o número de intervalos aumenta, `noIntervalo` torna-se intragável. Com dois intervalos, não é tão ruim:

```
inferior = noIntervalo(a, 0.0, 0.5)
superior = noIntervalo(a, 0.5, 1)
```

Mas com quatro intervalos, começa a ficar desconfortável.:

```
intervalo1 = noIntervalo(a, 0.0, 0.25)
intervalo2 = noIntervalo(a, 0.25, 0.5)
intervalo3 = noIntervalo(a, 0.5, 0.75)
intervalo4 = noIntervalo(a, 0.75, 1.0)
```

Existem aqui dois problemas. Um é que temos que criar novos nomes de variável para cada resultado. O outro é que temos que calcular os limites de cada intervalo.

Vamos resolver o segundo problema primeiro. Se o número de intervalos é `numeroDeIntervalos`, então a largura de cada intervalo é 1.0 / `numeroDeIntervalos`.

Vamos usar um laço (*loop*) para calcular a faixa, ou largura, de cada intervalo. A variável do *loop*, `i`, conta de 0 até `numeroDeIntervalos-1`:

```
larguraDoIntervalo = 1.0 / numeroDeIntervalos
for i in range(numeroDeIntervalos):
  inferior = i * larguraDoIntervalo
  superior = inferior + larguraDoIntervalo
  print "do" inferior, "ao", superior
```

Para calcular o limite inferior (`inferior`) de cada intervalo, nós multiplicamos a variável do *loop* (`i`) pela largura do intervalo (`larguraDoIntervalo`). O limite superior (`superior`) está exatamente uma "largura de intervalo" acima.

Com `numeroDeIntervalos = 8`, o resultado é:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
```

```
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

Você pode confirmar que cada intervalo tem a mesma largura, que eles não se sobrepõe, e que eles cobrem toda a faixa de valores de 0.0 a 1.0.

Agora, de volta ao primeiro problema. Nós precisamos de uma maneira de guardar oito inteiros, usando a váriavel do *loop* para indicar cada um destes inteiros. Você deve estar pensando, "Lista!"

Nós temos que criar a lista de intervalos fora do *loop*, porque queremos fazer isto apenas uma vez. Dentro do loop, nós vamos chamar `noIntervalo` repetidamente e atualizar o `i`-ésimo elemento da lista:

```
numeroDeIntervalos = 8
intervalos = [0] * numeroDeIntervalos
larguraDoIntervalo = 1.0 / numeroDeIntervalos
for i in range(numeroDeIntervalos):
  inferior = i * larguraDoIntervalo
  superior = inferior + larguraDoIntervalo
  intervalos[i] = noIntervalo(lista, inferior, superior)
print intervalos
```

Com uma lista de 1000 valores, este código vai produzir esta lista de quantidades de valores em cada intervalo:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

Esses números estão razoavelmente póximos de 125, o que era o que esperávamos. Pelo menos eles estão próximos o bastante para nos fazer acreditar que o gerador de número aleatórios está funcionando.

> *Como exercício, teste esta função com algumas listas longas, e veja se o número de valores em cada um dos intervalos tendem a uma distribuição nivelada.*

11.8 9.8 Uma solução em um só passo

Embora este programa funcione, ele não é tão eficiente quanto poderia ser. Toda vez que ele chama `noIntervalo`, ele percorre a lista inteira. Conforme o número de intervalos aumenta, a lista será percorrida um bocado de vezes.

Seria melhor fazer uma única passagem pela lista e calcular para cada valor o índice do intervalo ao qual o valor pertença. Então podemos incrementar o contador apropriado.

Na seção anterior, pegamos um índice, `i`, e o multiplicamos pela `larguraDoIntervalo` para encontrar o limite inferior daquele intervalo. Agora queremos pegar um valor entre 0.0 e 1.0 e encontrar o índice do intervalo ao qual ele se encaixa.

Já que este problema é o inverso do problema anterior, podemos imaginar que deveríamos dividir por `larguraDoIntervalo` em vez de multiplicar. Esta suposição está correta.

Já que `larguraDoIntervalo` = 1.0 / numeroDeIntervalos, dividir por `larguraDoIntervalo` é o mesmo que multiplicar por `numeroDeIntervalos`. Se multiplicarmos um número na faixa entre 0.0 e 1.0 por `numeroDeIntervalos`, obtemos um número na faixa entre 0.0 e `numeroDeIntervalos`. Se arredondarmos este número para baixo, ou seja, para o menor inteiro mais próximo, obtemos exatamente o que estamos procurando - o índice do intervalo:

```
numeroDeIntervalos = 8
intervalos = [0] * numeroDeIntervalos
for i in lista:
```

```
indice = int(i * numeroDeIntervalos)
intervalos[indice] = intervalos[indice] + 1
```

Usamos a função `int` para converter um número em ponto flutuante (*float*) para um inteiro.

Existe a possibilidade deste cálculo produzir um índice que esteja fora dos limites (seja negativo ou maior que `len(intervalos)-1`)?

Uma lista como `intervalos` que contém uma contagem do número de valores em cada intervalo é chamada de **histograma**.

> *Como exercício, escreva uma função chamada ‘‘histograma‘‘ que receba uma lista e um número de intervalos como argumentos e retorne um histograma com o número de intervalos solicitado.*

11.9 9.9 Glossário

tipo imutável (*immutable type*) Um tipo de elemento que não pode ser modificado. Atribuições a um elemento ou "fatiamento (slices)" XXX aos tipos imutáveis causarão erro.

tipo mutável (*mutable type*) Tipo de dados onde os elementos podem ser modificados. Todos os tipos mutáveis, são tipos compostos. Listas e dicionários são exemplos de tipos de dados mutáveis. String e tuplas não são.

tupla (*tuple*) Tipo sequencial similar as listas com exceção de que ele é imutável. Podem ser usadas Tuplas sempre que um tipo imutável for necessário, por exemplo uma "chave (key)" em um dicionário

Atribuição a tupla (*tuple assignment*) Atribuição a todos os elementos de uma tupla feita num único comando de atribução. A atribuição aos elementos ocorre em paralelo, e não em sequência, tornando esta operação útil para *swap*, ou troca recíproca de valores entre variáveis (ex: a,b=b,a).

determinístico (*deterministic*) Um programa que realiza a mesma coisa sempre que é executado.

pseudo aleatório (*pseudorandom*) Uma sequência de números que parecem ser aleatórios mas são na verdade o resultado de uma computação "determinística"

histograma (*histogram*) Uma lista de inteiros na qual cada elemento conta o número de vezes que algo acontece.

casamento de padrão XXX (*pattern matching*) Um programa desenvolvido que envolve identificar um teste padrão computacional familiar e copiar a solução para um problema similar.

CAPÍTULO **12**

Capítulo 10: Dicionários

Tópicos

Os tipos compostos que voce aprendeu - strings, listas e tuplas - utilizam inteiros como indices. Se voce tentar utilizar qualquer outro tipo como indice, voce receberá um erro.

Dicionários sao similiares a outros tipos compostos exceto por eles poderem user qualquer tipo imutavel de dados como indice. Como exemplo, nos criaremos um dicionário para traduzir palavras em Inglês para Espanhol. Para esse dicionário, os indices serão strings.

Uma maneira de criar um dicionario é comecando com um dicionário vazio e depois adiconando elementos. Um dicionário vazio é denotado assim *{}*:

```
>>> ing2esp = {}
>>> ing2esp['one'] = 'uno'
>>> ing2esp['two'] = 'dos'
```

A primeira atribuição cria um dicionario chamado `ing2esp`; as outras atribuições adicionam novos elementos para o dicionário. Nos podemos imprimir o valor corrente de um dicionario da maneira usual:

```
>>> print ing2esp
{'one': 'uno', 'two': 'dos'}
```

Os elementos de um dicionário aparecem em uma lista separada por vírgulas. Cada entrada contêm um indice e um valor separado por dois-pontos. Em um dicionário, os índices sao chamados de *chaves*, entao os elementos são chamados de *pares chave-valor*.

Outra maneira de criar dicionários é fornecendo uma lista de pares chaves-valor utilizando a mesma sintaxe da última saída.

```
>>> ing2esp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Se nos imprimirmos o valor de *ing2esp* novamente, nos teremos uma surpresa:

```
>>> print ing2esp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Os pares chave-valor não estão em ordem! Felizmente, não a motivos para se preocupar com a ordem, desde que os elementos do dicionário nunca sejam indexados com indices inteiros. Podemos usar as chaves para buscar os valores correspondentes:

```
>>> print ing2esp['two']
'dos'
```

A chave 'two' retornou o valor 'dos' mesmo pensando que retornaria o terceiro par chave-valor.

12.1 10.1 Operações dos Dicionários

O comando *del* remove um par chave-valor de um dicionário. Por exemplo, o dicionário abaixo contem os nomes de varias frutas e o numero de cada fruta em no estoque:

```
>>> inventario = {'abacaxis': 430, 'bananas': 312, 'laranjas': 525, 'peras': 217}
>>> print inventario
{'laranjas': 525, 'abacaxis': 430, 'peras': 217, 'bananas': 312}
```

Se alguem comprar todas as peras, podemos remover a entrada do dicionário:

```
>>> del inventario['peras']
>>> print inventario
{'laranjas': 525, 'abacaxis': 430, 'bananas': 312}
```

Ou se nós esperamos por mais peras em breve, nos podemos simplesmente trocar o valor associoado as peras:

```
>>> inventario['peras'] = 0
>>> print inventario
{'laranjas': 525, 'abacaxis': 430, 'peras': 0, 'bananas': 312}
```

A função *len* também funciona com dicionários; retornando o número de pares chave-valor:

```
>>> len(inventario)
4
```

12.2 10.2 Métodos dos Dicionários

Um *método* é parecido com uma função - possui parametros e retorna valores - mas a sintaxe é diferente. Por exemplo, o metodo `keys` recebe um dicionário e retorna uma lista com as chaves, mas em vez de usarmos a sintaxe de função `keys(ing2esp)`, nos usamos a sintaxe de método `ing2esp.keys()`:

```
>>> ing2esp.keys()
['one', 'three', 'two']
```

Dessa forma o ponto especifica o nome da função, `keys`, e o nome do objeto em que deve ser aplicada a função, `ing2esp`. Os parenteses indicam que esse método não possui parameteros.

Ao invés de chamarmos um método, dizemos que ele é *invocado*, nesse caso, nós podemos dizer que nós estamos invocando `keys` do objeto `ing2esp`.

O método `values` é parecido; retorna a lista de valores de um dicionário:

```
>>> ing2esp.values()
['uno', 'tres', 'dos']
```

O método `items` retorna os dois, na forma de uma lista de tuplas - cada tupla com um par chave-valor:

```
>>> ing2esp.items()
[('one','uno'), ('three','tres'), ('two','dos')]
```

A sintaxe fornece uma informação util. Os colchetes indicam que isso é uma lista. Os parentêses indicam que os elementos da lista são tuplas.

Se o método recebe de algum parâmetro, se utiliza a mesma sintaxe das funções. Por exemplo, o método `has_key` recebe uma chave e retorna verdadeiro (1) se a chave existe no dicionário:

```
>>> ing2esp.has_key('one')
True
>>> ing2esp.has_key('deux')
False
```

Se voce tentar chamar um método sem especificar em qual objeto, voce obterá um erro. Nesse caso, a mensagem de erro não é muito útil:

```
>>> has_key('one')
NameError: has_key
```

12.3 10.3 Aliasing (XXX) e Copiar

Uma vez que os dicionários são mutáveis, voce precisa saber sobre **Aliasing**. Sempre que duas variáveis referenciarem o mesmo objeto, quando uma é alterada, afeta a outra.

Se você quer modificar um dicionário e continuar com uma copia original, utilize o método **copy**. Por exemplo, **opposites** é um dicionário que contêm pares de antônimos:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposities
>>> copy = opposities.copy()
```

alias e **opposites** se referem ao mesmo objeto; **copy** se refere a um novo objeto igual ao dicionário **opposites**. Se você modificar o **alias**, **opposites** também será alterado.

```
>>> alias['right'] = 'left'
>>> opossites['right']
'left'
```

Se modificarmos **copy**, **opposites** não será modificado:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

12.4 10.4 Matrizes Esparsas

Na seção 8.14, nós usamos uma lista de listas para representar uma matriz. Essa é uma boa escolha se a matriz for principalmente de valores diferentes de zero, mas considerando uma matriz esparsa como essa:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Uma representação usando uma lista contem muitos zeros:

```
>>> matriz = [ [0,0,0,1,0],
               [0,0,0,0,0],
               [0,2,0,0,0],
               [0,0,0,0,0],
               [0,0,0,3,0] ]
```

Uma alternativa é usarmos um dicionário. Para as chaves, nós podemos usar tuplas que contêm os números da linha e a coluna. Abaixo uma representação em um diciónario da mesma matriz:

```
>>> matriz = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Nós precisamos apenas de três pares chave-valor, cada um sendo um elemento diferente de zero da matriz. Cada chave é uma tupla, e cada valor é um número inteiro.

Para acessarmos um elemento da matriz, nos utilizamos o operador []:

```
>>> matriz[0,3]
1
```

Note que a sintaxe da representação de um dicionário não é a mesma que a sintaxe usada pela representação pelas listas. Em vez de usarmos dois índices inteiros, nós usamos apenas um índice, que é uma tupla de inteiros.

Mas existe um problema. Se tentarmos buscar um elemento zero, obteremos um erro, pois não existe uma entrada no dicionário para a chave especificada:

```
>>> matriz[1,3]
KeyError: (1,3)
```

O método **get** resolve esse problema:

```
>>> matriz.get((0,3), 0)
1
```

O primeiro parâmetro é a chave; o segundo é o valor que **get** retornará caso não existe a chave no dicionário:

```
>>> matriz.get((1,3), 0)
0
```

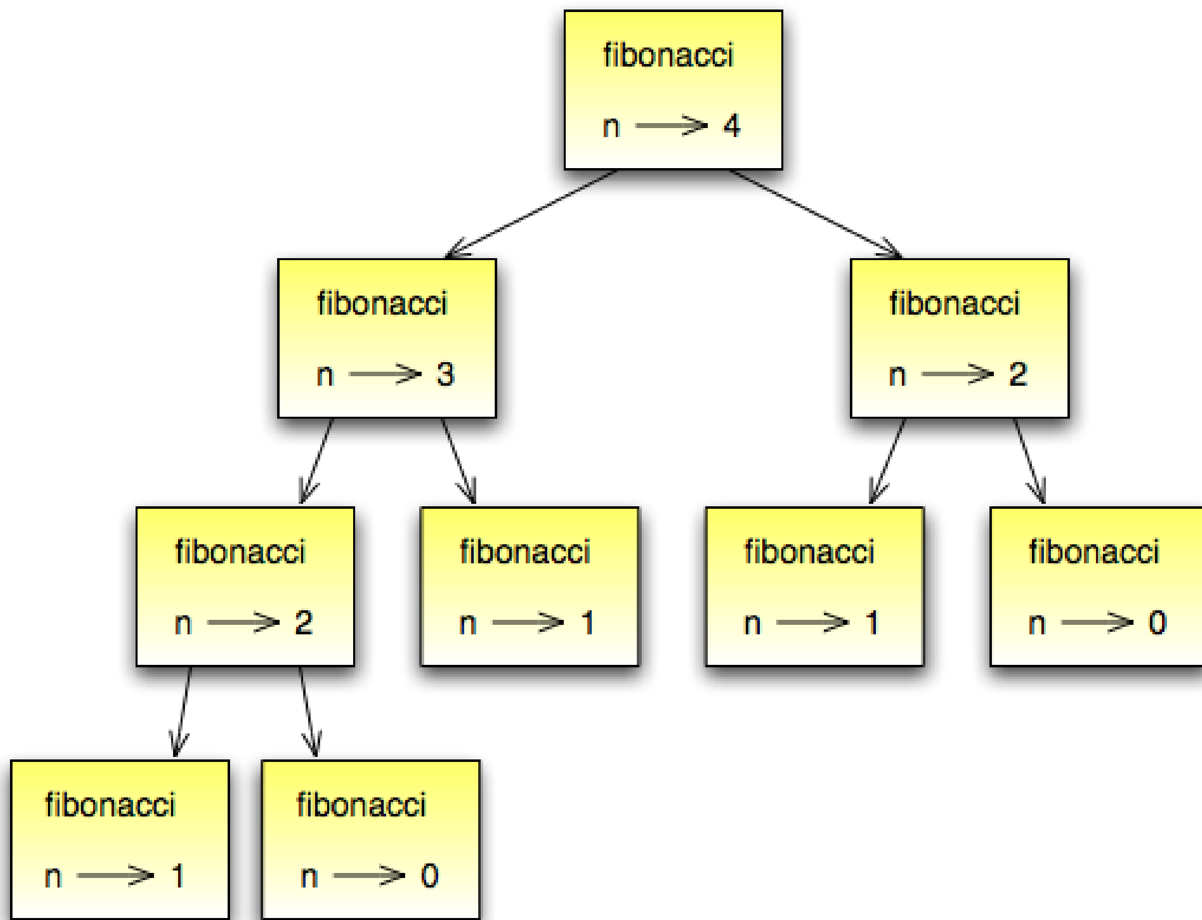
get definitivamente melhora a semântica e a sintaxe do acesso a matrizes esparsas.

12.5 10.5 Hint XXX

Se você brincou com a função **fibonacci** da seção 5.7, é provável que você notou que quanto maior o número passado para a função, mais tempo a função demora para executar. Além disso, o tempo da execução aumenta rapidamente.

Em uma das nossas máquinas, **fibonacci(20)** executa instantaneamente, **fibonacci(30)** demora cerca de um segundo, e **fibonacci(40)** demora uma eternidade.

Para entender o porque, considere o gráfico de chamadas para **fibonacci** com n=4:



O gráfico mostra a estrutura da função, com linhas conectando cada execução com a execução que a chamou. No topo do gráfico, **fibonacci** tem **n=4**, que chama **fibonacci** com **n=3** e **n=2**. Em seguida, **fibonacci** com **n=3** chama **fibonacci** com **n=2** e **n=1**. E assim por diante.

Conte quantas vezes **fibonacci(0)** e **fibonacci(1)** são chamadas. Essa é uma solução ineficiente para o problema, e torna-se pior quando o parâmetro recebido é um número maior.

Uma boa solução é guardar os valores que já foram calculados armazenando-os em um dicionário. Um valor previamente calculado que é guardado para ser utilizado mais tarde é chamado de **hint**. Abaixo uma implementação de **fibonacci** usando **hints**:

```
>>> previous = {0:1, 1:1}
>>> def fibonacci(n):
        if previous.has_key(n):
                return previous[n]
        else:
                newValue = fibonacci(n-1) + fibonacci(n-2)
                previous[n] = newValue
                return newValue
```

O dicionário chamado **previous** guarda os números de Fibonacci que nós ja conhecemos. Ele começa com apenas dois pares: 0 possui 1; e 1 possui 1.

Sempre que **fibonacci** é chamada, ela verifica o dicionário para determinar se ele já possui o resultado. Se o resultado estiver ali, a função pode retornar imediatamente sempre precisar fazer mais chamadas recursivas. Se o resultado não estiver ali, ele é calculado no **newValue**. O valor de **newValue** é adicionado no dicionário antes da função retornar.

Usando essa versão de **fibonacci**, nossa máquina consegue calcular **fibonacci(40)** em um piscar de olhos. Mas quando tentamos calcular **fibonacci(50)**, nós veremos um problema diferente:

```
>>> fibonacci(50)
OverflowError: integer addition
```

A resposta, que você verá em um minuto, é 20.365.011.074. O problema é que esse número é muito grande para guardarmos como um inteiro do Python [1]. Isso é **overflow**. Felizmente, esse problema tem uma solução simples.

12.6 10.6 Inteiros Longos

Python possui um tipo chamado **long int** que permite trabalharmos com qualquer tamanho de inteiros. Existem duas maneiras de criarmos um valor **long int**. A primeira é escrever um inteiro seguido de um L no final:

```
>>> type(1L)
<type 'long int'>
```

A outra maneira é usarmos a função **long** que converte um valor para um **long int**. **long** pode receber qualquer valor númerico e até mesmo uma string de digitos:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

Todas as operações matemáticas funcionam com **long int** s, então não precisamos modificar muito para adaptar **fibonacci**:

```
>>> previous = {0: 1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

Somente trocando os valores iniciais de **previous**, conseguimos mudar o comportamento da **fibonacci**. Os dois primeiros numeros da sequência são **long ints**, então todos os números subsequentes da sequência também serão.

> Como exercício, converta `fatorial` para produzir um inteiro longo como resultado.

12.7 10.7 Contando Letras

No capítulo 7, escrevemos uma função que contava o número de ocorrências de uma letra em uma string. A versão mais comum desse problema é fazer um histograma das letras da string, ou seja, quantas vezes cada letra aparece na string.

Um histograma pode ser util para comprimir um arquivo de texto. Pois diferentes letras aparecem com diferentes frequências, podemos comprimir um arquivo usando pequenos códigos para letras comuns e longos códigos para letras que aparecem em menor frequência.

Dicionários fornecem uma maneira elegante de gerar um histograma:

[1] N.T. A partir do Python 2. XXX este erro não ocorre mais, pois em caso de sobrecarga o valor inteiro é automaticamente promovido para o tipo long.

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...     letterCounts[letter] = letterCounts.get(letter,0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Começamos com um dicionário vazio. Para cada letra da string, achamos o contador (possivelmente zero) e o incrementamos. No final, o dicionário contem pares de letras e as suas frequências.

É mais atraente mostrarmos o histograma na ordem alfabética. Podemos fazer isso com os métodos **items** e **sort**:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Você ja tinha visto o método **items** antes, mas **sort** é o primeiro método que você se depara para aplicar em listas. Existem muitos outros métodos de listas, incluindo **append**, **extend**, e **reverse**. Consulte a documentação do Python para maiores detalhes.

12.8 10.8 Glossário

dicionário (*dictionary*) Uma coleção de pares de chaves-valores que são mapeados pelas chaves, para se obter os valores. As chaves podem ser qualquer tipo de dados imutavel, e os valores podem ser de qualquer tipo.

chave (*key*) Um valor que é usado para buscar uma entrada em um dicionário.

par chave-valor (*key-value pair*) Um dos itens de um dicionário.

método (*method*) Um tipo de função que é chamada com uma sintaxe diferente e invocada no contexto de um objeto.

invocar (*invoke*) Chamar um método.

hint O armazenamento temporário de um valor pré-computado para evitar a computação redundante.

overflow Um resultado numérico que é muito grande para ser representado no formato numérico.

Capítulo 11: Arquivos e exceções

Tópicos

13.1 Arquivos e exceções

Durante a execução de um programa, seus dados ficam na memória. Quando o programa termina, ou o computador é desligado, os dados na memória desaparecem. Para armazenar os dados permanentemente, você tem que colocá-los em um **arquivo**. Arquivos usualmente são guardados em um disco rígido (HD), num disquete ou em um CD-ROM.

Quando existe um número muito grande de arquivos, eles muitas vezes são organizados dentro de **diretórios** (também chamados de ?pastas? ou ainda ?*folders*?). Cada arquivo é identificado por um nome único, ou uma combinação de um nome de arquivo com um nome de diretório.

Lendo e escrevendo em arquivos, os programas podem trocar informações uns com os outros e gerar formatos imprimíveis como PDF.

Trabalhar com arquivos é muito parecido com trabalhar com livros. Para utilizar um livro, você tem que abrí-lo. Quando você termina, você tem que fechá-lo. Enquanto o livro estiver aberto, você pode tanto lê-lo quanto escrever nele. Em qualquer caso, você sabe onde você está situado no livro. Na maioria das vezes, você lê o livro inteiro em sua ordem natural, mas você também pode saltar através de alguns trechos (skip around).

Tudo isso se aplica do mesmo modo a arquivos. Para abrir um arquivo, você especifica o nome dele e indica o que você quer, seja ler ou escrever (gravar).

Abrir um arquivo cria um objeto arquivo. Neste exemplo, a variável `f` se referencia ao novo objeto arquivo.

```
>>> f = open("teste.dat", "w")
>>> print f
<open file "teste.dat", mode "w" at fe820>
```

A função `open` recebe dois argumentos. O primeiro é o nome do arquivo, e o segundo é o modo. Modo ?w? significa que estamos abrindo o arquivo para gravação (?*write*?, escrever).

Se não existir nenhum arquivo de nome `teste.dat`, ele será criado. Se já existir um, ele será substituído pelo arquivo que estamos gravando (ou escrevendo).

Quando executamos um comando `print` sobre o objeto arquivo, visualizamos o nome do arquivo, o modo e a localização do objeto na memória.

Para colocar dados dentro do arquivo, invocamos o método `write` do objeto arquivo:

```
>>> f.write("Agora é hora")
>>> f.write("de fechar o arquivo")
```

Fechar o arquivo diz ao sistema que terminamos de escrever (gravar) e que o arquivo está livre para ser lido:

```
>>> f.close()
```

Agora podemos abrir o arquivo de novo, desta vez para leitura, e ler o seu conteúdo para uma string. Desta vez, o argumento modo é ?r? para leitura (?reading?):

```
>>> f = open("teste.dat", "r")
```

Se tentarmos abrir um arquivo que não existe, temos um erro:

```
>>> f = open("teste.cat", "r")
IOError: [Errno 2] No such file or directory: 'teste.cat'
```

Sem nenhuma surpresa, o método `read` lê dados do arquivo. Sem argumentos, ele lê todo o conteúdo do arquivo:

```
>>> texto = f.read()
>>> print texto
Agora é horade fechar o arquivo
```

Não existe espaço entre ?hora? e ?de? porque nós não gravamos um espaço entre as strings.

`read` também pode receber um argumento que indica quantos caracteres ler:

```
>>> f = open("teste.dat", "r")
>>> print f.read(9)
Agora é h
```

Se não houver caracteres suficientes no arquivo, `read` retorna os caracteres restantes. Quando chegamos ao final do arquivo, `read` retorna a string vazia:

```
>>> print f.read(1000006)
orade fechar o arquivo
>>> print f.read()

>>>
```

A função seguinte, copia um arquivo, lendo e gravando até cinqüenta caracteres de uma vez. O primeiro argumento é o nome do arquivo original; o segundo é o nome do novo arquivo:

```
def copiaArquivo(velhoArquivo, novoArquivo):
  f1 = open(velhoArquivo, "r")
  f2 = open(novoArquivo, "w")
  while 1:
```

```
        texto = f1.read(50)
        if texto == "":
            break
        f2.write(texto)
    f1.close()
    f2.close()
    return
```

A comando `break` é novo. O que ele faz é saltar a execução para fora do loop; o fluxo de execução passa para o primeiro comando depois do loop.

Neste exemplo, o loop `while` é infinito porque o valor 1 é sempre verdadeiro. O único modo de sair do loop é executando o `break`, o que ocorre quando `texto` é a string vazia, o que ocorre quando alcançamos o fim do arquivo.

13.2 11.1 Arquivos texto

Um arquivo texto é um arquivo que contém caracteres imprimíveis e espaços, organizados dentro de linhas separadas por caracteres de nova linha. Já que Pyhton é especialmente projetado para processar arquivos texto, ele oferece métodos que tornam esta tarefa mais fácil.

Para demonstrar, vamos criar um arquivo texto com três linhas de texto separadas por caracteres de nova linha:

```
>>> f = open("teste.dat", "w")
>>> f.write("linha um\nlinha dois\nlinha três\n")
>>> f.close()
```

O método `readline` lê todos os caracteres até, e incluindo, o próximo caractere de nova linha:

```
>>> f = open("teste.dat", "r")
>>> print f.readline()
linha um

>>>
```

readlines retorna todas as linhas restantes como uma lista de strings:

```
>>> print f.readlines()
['linha dois\012', 'linha três\012']
```

Neste caso, a saída está em formado de lista, o que significa que as strings aparecem entre aspas e o caractere de nova linha aparece como a seqüência de escape `012`.

No fim do arquivo, `readline` retorna a string vazia e `readlines` retorna a lista vazia:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

A seguir temos um exemplo de um programa de processamento de linhas. `filtraArquivo` faz uma cópia de `velhoArquivo`, omitindo quaisquer linhas que comecem por `#`:

```
def filtraArquivo(velhoArquivo, novoArquivo):
    f1 = open(velhoArquivo, "r")
    f2 = open(novoArquivo, "w")
    while 1:
        texto = f1.readline()
        if texto == "":
```

```
            break
        if texto[0] == '#':
            continue
        f2.write(texto)
    f1.close()
    f2.close()
    return
```

O comando `continue` termina a iteração corrente do loop, mas continua iterando o loop. O fluxo de execução passa para o topo do loop, checa a condição e prossegue conforme o caso.

Assim, se `texto` for a string vazia, o loop termina. Se o primeiro caractere de texto for o jogo da velha (? # ?), o fluxo de execução passa para o topo do loop. Somente se ambas as condições falharem é que `texto` será copiado para dentro do novo arquivo.

13.3 11.2 Gravando variáveis

O argumento de `write` tem que ser uma string, assim se quisermos colocar outros valores em um arquivo, temos de convertê-los para strings primeiro. A maneira mais fácil de fazer isso é com a função `str`:

```
>>> x = 52
>>> f.write(str(x))
```

Uma alternativa é usar o **operador de formatação** `%`. Quando aplicado a inteiros, `%` é o operador módulo. Mas quando o primeiro operador é uma string, `%` é o operador de formatação.

O primeiro operando é a **string de formatação**, e o segundo operando é uma tupla de expressões. O resultado é uma string que contém os valores das expressões, formatadas de acordo com a string de formatação.

Num exemplo simples, a **seqüência de formatação** "??%d??" significa que a primeira expressão na tupla deve ser formatada como um inteiro. Aqui a letra *d* representa ?decimal?.

```
>>> carros = 52
>>> "%d" % carros
'52'
```

O resultado é a string ?52?, que não deve ser confundida com o valor inteiro 52.

Uma seqüência de formatação pode aparecer em qualquer lugar na string de formatação, assim, podemos embutir um valor em uma seqüência:

```
>>> carros = 52
>>> "Em julho vendemos %d carros." % carros
'Em julho vendemos 52 carros.'
```

A seqüência de formatação "`%f`" formata o próximo item da tupla como um número em ponto flutuante, e "%s" formata o próximo como uma string:

```
>>> "Em %d dias fizemos %f milhões %s." % (34,6.1,'reais')
'Em 34 dias fizemos 6.100000 milhões de reais.'
```

Por padrão, o formato de ponto flutuante exibe seis casas decimais.

O número de expressões na tupla tem que ser igual ao número de seqüências de formatação na string. Além disso, os tipos das expressões têm que iguais aos da seqüência de formatação:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
```

```
>>> "%d" % 'reais'
TypeError: illegal argument type for built-in operation
```

No primeiro exemplo, não existem expressões suficientes; no segundo, a expressão é do tipo errado.

Para um controle maior na formatação de números, podemos especificar o número de dígitos como parte da seqüência de formatação:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6,100000'
```

O número depois do sinal de porcentagem é o número mínimo de espaços que o valor ocupará. Se o valor fornecido tiver um número menor de dígitos, espaços em branco serão adicionados antes para preencher o restante. Se o número de espaços for negativo, os espaços serão adicionados depois:

```
>>> "%-6d" % 62
'62    '
```

Para números em ponto-flutuante, também podemos especificar o número de dígitos depois da vírgula:

```
>>> "%12.2f" % 6.1
'        6.10'
```

Neste exemplo, o resultado reserva 12 espaços e inclui dois dígitos depois da vírgula. Esta formatação é útil para exibir valores monetários com os centavos alinhados.

Por exemplo, imagine um dicionário que contém nomes de estudantes como chaves e salários-hora como valores. Aqui está uma função que imprime o conteúdo do dicionário como um relatório formatado:

```
def relatorio(salarios):
  estudantes = salarios.keys()
  estudantes.sort()
  for estudante in estudantes:
      print "%-20s %12.02f" % (estudante, salarios[estudante])
```

Para testar esta função, criaremos um pequeno dicionário e imprimiremos o conteúdo:

```
>>> salarios = {'maria': 6.23, 'joão': 5.45, 'josué': 4.25}
>>> relatorio(salarios)
joão                         5.45
josué                        4.25
maria                        6.23
```

Controlando a largura de cada valor, podemos garantir que as colunas ficarão alinhadas, desde que os nomes contenham menos que vinte e um caracteres e os salários sejam menores do que um bilhão de reais por hora.

13.4 11.3 Diretórios

Quando você cria um novo arquivo abrindo-o e escrevendo nele, o novo arquivo fica no diretório corrente (seja lá onde for que você esteja quando rodar o programa). Do mesmo modo, quando você abre um arquivo para leitura, Python procura por ele no diretório corrente.

Se você quiser abrir um arquivo que esteja em algum outro lugar, você tem que especificar o **caminho** (*path*) para o arquivo, o qual é o nome do diretório (ou folder) onde o arquivo está localizado:

```
>>> f = open("/usr/share/dict/words", "r")
>>> print f.readline()
Aarhus
```

Este exemplo abre um arquivo chamado `words` que reside em um diretório de nome `dict`, o qual reside em `share`, o qual reside em `usr`, o qual reside no diretório de mais alto nível do sistema, chamado `/`.

Você não pode usar `/` como parte do nome de um arquivo; ela é um caractere reservado como um delimitador entre nomes de diretórios e nomes de arquivos.

O arquivo `/usr/share/dict/words` contém uma lista de palavras em ordem alfabética, na qual a primeira palavra é o nome de uma universidade Dinamarquesa.

13.5 11.4 Pickling

Para colocar valores em um arquivo, você tem que convertê-los para strings. Você já viu como fazer isto com `str`:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

O problema é que quando você lê de volta o valor, você tem uma string. O Tipo original da informação foi perdido. De fato, você não pode sequer dizer onde começa um valor e termina outro:

```
>>> f.readline()
?12.3[1, 2, 3]?
```

A solução é o pickling, assim chamado porque ?preserva? estruturas de dados. O módulo `pickel` contém os comandos necessários. Para usá-lo, importe `pickle` e então abra o arquivo da maneira usual:

```
>>> import pickle
>>> f = open(?test.pck?, ?w?)
```

Para armazenar uma estrutura de dados, use o método `dump` e então feche o arquivo do modo usual:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Então, podemos abrir o arquivo para leitura e carregar as estruturas de dados que foram descarregadas (dumped):

```
>>> f = open(?test.pck?, ?r?)
>>> x = pickle.load(f)
>>> x
12,3
>>> type(x)
<type ?float?>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type ?list?>
```

Cada vez que invocamos `load`, obtemos um único valor do arquivo, completo com seu tipo original.

13.6 11.5 Exceções

Whenever que um erro em tempo de execução acontece, ele gera uma exceção. Usualmente, o programa pára e Python exibe uma mensagem de erro.

Por exemplo, dividir por zero gera uma exceção:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

Do mesmo modo, acessar um item de lista inexistente:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

Ou acessar uma chave que não está em um dicionário:

```
>>> b = {}
>>> print b[?what?]
KeyError: what
```

Em cada caso, a mensagem de erro tem duas partes: o tipo do erro antes dos dois pontos, e especificidades do erro depois dos dois pontos. Normalmente Python também exibe um ?*traceback*? de onde estava a execução do programa, mas nós temos omitido esta parte nos exemplos.

Às vezes queremos executar uma operação que pode causar uma exceção, mas não queremos que o programa pare. Nós podemos tratar a exceção usando as instruções `try` e `except`.

Por exemplo, podemos pedir ao usuário um nome de arquivo e então tentar abrí-lo. Se o arquivo não existe, não queremos que o programa trave; queremos tratar a exceção:

```
nomedoarquivo = raw_input(?Entre com o nome do arquivo: ?)
try:
  f = open (nomedoarquivo, ?r?)
except:
  print ?Não existe arquivo chamado?, nomedoarquivo
```

A instrução `try` executa os comandos do primeiro bloco. Se não ocorrerem exceções, ele ignora a instrução `except`. Se qualquer exceção acontece, ele executa os comandos do ramo `except` e continua.

Podemos encapsular esta habilidade numa função: existe toma um nome de arquivo e retorna verdadeiro se o arquivo existe e falso se não existe:

```
def existe(nomedoarquivo)
  try:
    f = open(nomedoarquivo)
    f.close()
    return 1
  except:
    return 0
```

Você pode usar múltiplos blocos `except` para tratar diferentes tipos de exceções. O Manual de Referência de Python (*Python Reference Manual*) tem os detalhes.

Se o seu programa detecta uma condição de erro, você pode fazê-lo lançar uma exceção. Aqui está um exemplo que toma uma entrada do usuário e testa se o valor é 17. Supondo que 17 não seja uma entrada válida por uma razão qualquer, nós lançamos uma exceção.

```
def entraNumero():
  x = input (?Escolha um número: ?)
  if x == 17:
    raise ?ErroNumeroRuim?, ?17 é um número ruim?
  return x
```

O comando `raise` toma dois argumentos: o tipo da exceção e informações específicas sobre o erro. `ErroNumeroRuim` é um novo tipo de exceção que nós inventamos para esta aplicação.

Se a função que chamou `entraNumero` trata o erro, então o programa pode continuar; de outro modo, Pyhton exibe uma mensagem de erro e sai:

```
>>> entraNumero()
Escolha um número: 17
ErroNumeroRuim: 17 é um número ruim
```

A mensagem de erro inclui o tipo da exceção e a informação adicional que você forneceu.

> Como um exercício, escreva uma função que use `entraNumero` para pegar um número do teclado e que trate a exceção `ErroNumeroRuim`.

13.7 11.6 Glossário

arquivo (*file*) Uma entidade nomeada, usualmente armazenada em um disco rígido (HD), disquete ou CD-ROM, que contém uma seqüência de caracteres.

diretório (*directory*) Uma coleção nomeada de arquivos, também chamado de pasta ou folder.

caminho (*path*) Uma seqüência de nomes de diretórios que especifica a exata localização de um arquivo.

arquivo texto (*text file*) Um arquivo que contém caracteres organizados em linhas separadas por caracteres de nova linha.

comando break (*break statement*) Um comando que força a atual iteração de um loop a terminar. O fluxo de execução vai para o topo do loop, testa a condição e prossegue conforme o caso.

Capítulo 12: Classes e objetos

Tópicos

14.1 12.1 Tipos compostos definidos pelo usuário

Depois de usarmos alguns tipos nativos do Python, estamos prontos para criar um tipo de dados: o `Ponto`.

Considere o conceito matemático de um ponto. Em duas dimensões, um ponto é um par de números (coordenadas) que são tratadas coletivamente como um objeto simples. Na notação matemática, pontos são freqüentemente escritos entre parênteses com vírgula separando as coordenadas. Por exemplo, (0, 0) representa a origem, e (x, y) representa o ponto x unidades à direita, e y unidades acima da origem.

Uma maneira natural para representar um ponto em Python, é com dois valores numéricos em ponto flutuante. A questão, então, é como agrupar estes dois valores em um objeto composto. A maneira rápida e rasteira é usar uma lista ou uma tupla, e para algumas aplicações, esso pode ser a melhor escolha [1].

Uma alternativa é definir um novo tipo composto, também chamado uma **classe**. Esta abordagem envolve um pouco mais de esforço, mas ela tem vantagens que logo ficarão evidentes.

Eis a definição de uma classe:

[1] **N.T.**: A linguagem Python também incorpora um tipo nativo `complex` que representa números complexos. Uma instância de `complex`, como `a=3+5j` possui dois valores de ponto flutuante em seus atributos `a.real` e `a.imag`, e pode ser utilizada para armazenar pontos em um espaço bi-dimensional.

```
class Ponto:
  pass
```

Definições de classes podem aparecer em qualquer parte de um programa, mas elas costuma ficar próximas do começo do programa (após os comandos `import`). As regras de sintaxe para a definição de classes são as mesmas de outros comandos compostos (veja Seção 4.4).

A definição acima cria uma nova classe chamada `Ponto`. O comando `pass` não tem nenhum efeito; aqui ele é necessário porque um comando composto precisa ter algo no seu corpo.

Quando criamos a classe `Ponto`, criamos um novo tipo de dado, também chamado `Ponto`. Os membros deste novo tipo são chamados **instâncias** deste tipo ou **objetos**. Criar uma nova instância é **instanciar**. Para instanciar o objeto `Ponto`, invocamos a função (adivinhou?) `Ponto`:

```
final = Ponto()
```

A variável `final` agora contém uma referência a um novo objeto da classe `Ponto`. Uma função como `Ponto`, que cria novos objetos, é chamada **construtor**.

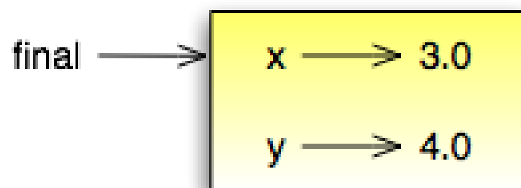
14.2 12.2 Atributos

Podemos adicionar novos dados em uma instância usando a notação de ponto (*dot notation*):

```
>>> final.x = 3.0
>>> final.y = 4.0
```

Esta sintaxe é similar à sintaxe para acessar uma variável de um módulo, como `math.pi` ou `string.uppercase`. Neste caso, porém, estamos acessando um item de dado de uma instância. Estes itens são chamados **atributos**.

O seguinte diagrama de estado mostra o resultado destas atribuições:



A variável `final` refere a um objeto `Ponto`, que contém dois atributos. Cada atributo faz referência a um número em ponto flutuante.

Podemos ler o valor de um atributo usando a mesma sintaxe:

```
>>> print final.y
4.0
>>> x = final.x
>>> print x
3.0
```

A expressão `final.x` significa, "Vá ao objeto `final` e pegue o valor de `x`". Neste caso, atribuímos este valor a uma variável cujo nome é 'x'. Não há conflito entre a variável `x` e o atributo `x`. O propósito da notação `objeto.atributo` é identificar a qual variável você está fazendo referência de forma que não é ambíguo.

Você pode usar a notação `objeto.atributo` como parte de qualquer expressão; assim os seguintes comandos são válidos:

```
print '(' + str(final.x) + ', ' + str(final.y) + ')'
distAoQuadrado = final.x * final.x + final.y * final.y
```

A primeira linha imprime (3.0, 4.0); a segunda linha calcula o valor 25.0.

É tentador imprimir o valor do próprio objeto `final`:

```
>>> print final
<__main__.Ponto instance at 80f8e70>
```

O resultado indica que `final` é uma instância da classe `Ponto` e foi definida no prgrama principal: `__main__`. 80f8e70 é o identificador único deste objeto, escrito em hexadecimal (base 16). Esta não é provavelmente a forma mais informativa para mostrar um objeto `Ponto`. Logo você irá ver como mudar isso.

> Como exercício, crie e imprima um objeto Ponto, e então use `id` para imprimir o identificador único do objeto. Traduza a forma hexadecimal para a forma decimal e confirme se são compatíveis.

14.3 12.3 Instâncias como parâmetros

Você pode passar uma instância como um parâmetro da forma usual. Por exemplo:

```
def mostrarPonto(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

A função `mostrarPonto` pega o ponto (p) como um argumento e mostra-o no formato padrão. Se você chamar `mostrarPonto(final)`, a saída será (3.0, 4.0).

> Como um exercício, re-escreva a função distância da Seção 5.2 para receber dois pontos como parâmetros, ao invés de quatro números.

14.4 12.4 O significado de "mesmo"

O significado da palavra "mesmo" parece perfeitamente claro até que você pense a respeito, e então você percebe que há mais nesta palavra do que você esperava.

Por exemplo, se você diz "Cris e eu temos o mesmo carro", você está dizendo que o carro de Cris e o seu são do mesmo fabricante e modelo, mas são dois carros diferentes. Se você disser "Cris e eu temos a mesma mãe", você está dizendo que a mãe de Cris e a sua, são a mesma pessoa [2]. Portanto a idéia de 'semelhança' é diferente dependendo do contexto.

Quando falamos de objetos, há uma ambigüidade similar. Por exemplo, se dois Pontos forem os mesmos, isto quer dizer que eles contêm os mesmos dados (coordenadas) ou que são realmente o "mesmo" objeto?

Para verificar se duas referências se referem ao 'mesmo' objeto, use o operador '==' [3]. Por exemplo:

```
>>> p1 = Ponto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Ponto()
```

[2] Nem todos os idiomas têm este problema. Por exemplo, em alemão há palavras diferentes para diferentes sentidos de "mesmo". "Mesmo carro" nesse contexto seria "gleiche Auto", e "mesma mãe" seria "selbe Mutter".

[3] XXX LR: Eu não diria que devemos usar == para verificar se dois objetos são o mesmo. Isto é uma falha do livro que talvez se origine no original que falava de Java. Em Python o operador is faz o mesmo que o == de Java: compara referências, e portanto serve para determinar se duas variáveis apontam para o mesmo objeto. No entanto, a o código acima está correto porque em Python a implemetação default de == (método __eq__) é comparar o id das instâncias, porém as classes list e dict, por exemplo, implementam __eq__ comparando os valores contidos (ex.: isto retorna True: l1 = [1,2,3]; l2 = [1,2,3]; l1 == l2).

```
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
False
```

Mesmo que p1 e p2 contenham as mesmas coordenadas, os dois não representam o mesmo objeto. Se atribuirmos p1 a p2, então as duas variáveis são pseudônimos do mesmo objeto.

```
>>> p2 = p1
>>> p1 == p2
True
```

Este tipo de igualdade é chamado de igualdade rasa porque ela compara somente as referências e não o conteúdo dos objetos.

Para comparar o conteúdo dos objetos – igualdade profunda – podemos escrever uma função chamada mesmoPonto:

```
def mesmoPonto(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```

Agora se criarmos dois diferentes objetos que contém os mesmos dados, podemos usar mesmoPonto para verificar se eles representam o mesmo ponto.

```
>>> p1 = Ponto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Ponto()
>>> p2.x = 3
>>> p2.y = 4
>>> mesmoPonto(p1, p2)
True
```

É claro, se as duas variáveis referirem ao mesmo objeto, elas têm igualdade rasa e igualdade profunda.

14.5 12.5 Retângulos

Digamos que desejemos uma classe para representar um retângulo. A questão é, qual informação temos de prover para especificar um retângulo? Para manter as coisas simples, assuma que o retângulo é orientado verticalmente ou horizontalmente, nunca em um ângulo.

Há algumas possibilidades: poderíamos especificar o centro do retângulo (duas coordenadas) e seu tamanho (largura e altura); ou poderíamos especificar um dos lados e o tamanho; ou poderíamos especificar dois lados opostos. A escolha convencional é especificar o canto superior esquerdo do retângulo e o tamanho.

Novamente, vamos definir uma nova classe:

```
class Rectangle:
  pass
```

E instanciá-la:

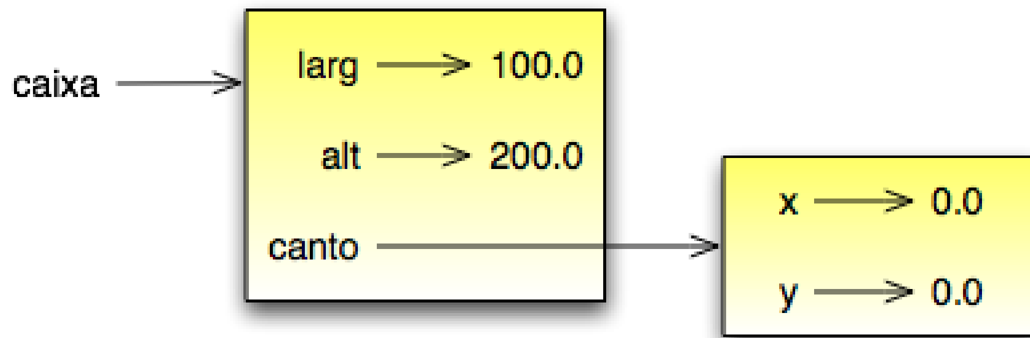
```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

Este código cria um novo objeto Retângulo com dois atributos ponto-flutuante. Para especificar o canto superior esquerdo, podemos embutir um objeto dentro de um objeto!

```
box.corner = Ponto()
box.corner.x = 0.0;
box.corner.y = 0.0;
```

A expressão box.corner.x significa, "vá ao objeto referenciado por 'box' e selecione o atributo 'corner'; então vá ao objeto 'corner' e deste, selecione o atributo de nome 'x'".

A figura mostra o estado deste objeto:



14.6 12.6 Instancias como valores retornados

Funções podem retornar instâncias. Por exemplo, findCenter pega um Retângulo como um argumento e retorna um Ponto que contem as coordenadas do centro do retângulo:

```
def findCenter(box):
      p = Ponto()
      p.x = box.corner.x + box.width/2.0
      p.y = box.corner.y + box.height/2.0
```

Para chamar esta função, passe 'box' como um argumento e coloque o resultado em uma variável.

```
>>> center = findCenter(box)
>>> print mostrarPonto(center)
(50.0, 100.0)
```

14.7 12.7 Objetos são mutáveis

Podemos mudar o estado de um objeto fazendo uma atribuição a um dos seus atributos. Por exemplo, para mudar o tamanho de um retângulo sem mudar sua posição, podemos modificar os valores de sua largura e altura. Veja:

```
box.width = box.width + 50
box.height = box.height + 100
```

Poderíamos encapsular este código em um método e generaliza-lo para aumentar o tamanho deste retângulo em qualquer medida:

```
def growRect(box, dwidth, dheight) :
      box.width = box.width + dwidth
      box.height = box.height + dheight
```

As variáveis dwidth e dheight indicam em quanto vamos aumentar o tamanho do retângulo em cada direção. Chamando este método, teríamos o mesmo efeito.

Por exemplo, poderíamos criar um novo Retângulo com o nome de 'bob' e passar este nome para o método growRect:

```
>>> bob = Rectangle()
>>> bob.width = 100.00
>>> bob.height = 200.00
>>> bob.corner.x = 0.0;
>>> bob.corner.y = 0.0;
>>> growRect(bob, 50, 100)
```

Enquanto growRect está sendo executado, o parâmetro 'box' é um alias (apelido) para 'bob'. Qualquer mudança feita em 'box', também irá afetar 'bob'.

Como exercício, escreva uma function (método) com o nome de moveRect que pega um Rectangle e dois parâmetros com o nome de 'dx' e 'dy'. Esta função deverá mudar a localização do retângulo através da adição de 'dx' à coordenada 'x' e da adição de 'dy' à coordenada 'y'.

14.8 12.8 Copiando

Ao usar 'alias' - como fizemos na seção anterior - podemos tornar o programa um pouco difícil de ler ou entender, pois as mudanças feitas em um local, podem afetar inesperadamente um outro objeto. E pode se tornar difícil de encontrar todas as variáveis que podem afetar um dado objeto.

Copiar um objeto é freqüentemente uma alternativa ao 'alias'. O modulo 'copy' contém uma função chamada 'copy' que duplica um qualquer objeto. Veja:

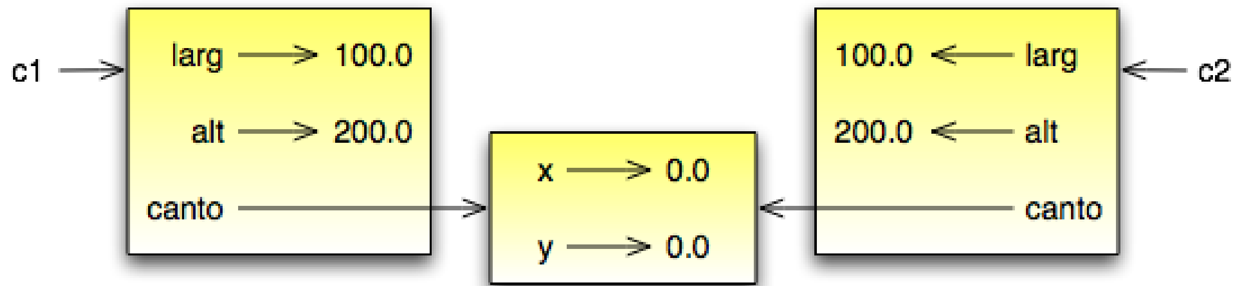
```
>>> import copy
>>> p1 = Ponto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
0
>>> mesmoPonto(p1, p2)
1
```

Uma vez que importamos o modulo 'copy', podemos usar o método 'copy' para criar um outro 'Ponto'. p1 e p2 não representam o mesmo ponto, mas eles contem os mesmo dados.

Para copiar um simples objeto como um 'Ponto', que não contem nenhum objeto embutido, 'copy' é suficiente. Isto eh chamado 'shallow' copia.

Mas para um objeto como um 'Rectangle', que contem uma referencia para um 'Ponto', o método 'copy' não irá executar corretamente a copia. Ele irá copiar a referencia para o objeto 'Ponto', portanto o que acontece aqui é que os dois Rectangle (o novo e o antigo) irão fazer referencia a um simples 'Ponto'.

Em outras palavras, se criarmos um 'box', c1, utilizando a forma usual, e depois fazer uma copia, c2, usando o método 'copy', o diagrama de estado resultante ficará assim:



o resultado não será o que esperamos. Neste caso, invocando 'growRect' em um dos retângulos (c1), isto não irá afetar o outro retângulo (c2, neste exemplo). Mas se usarmos o método 'moveRect' em qualquer um deles, isto irá inevitavelmente afetar o outro. Este comportamento é confuso e propenso a erros!

Mas felizmente o modulo 'copy' contem um método chamado 'deepcopy' que copia não somente o objeto, mas também copia todo e qualquer objeto 'embutido' neste objeto. Por isto, você não ficará surpreso porque este método chama-se 'deepcopy' (copia profunda) não é? Veja como funciona:

```
>>> c2 = copy.deepcopy(c1)
```

Agora, c1 e c2 são objetos completamente separados.

Podemos usar 'deepcopy' para re-escrever 'growRect' sendo que ao invés de modificar um Rectangle existente, ele cria um novo que tem a mesma localização do outro, mas com novas dimensões:

```
def growRect(box, dwidth, dheight):
  import copy
  newBox = copy.deepcopy(box)
  newBox.width = newBox.width + dwidth
  newBox.height = newBox.height + dheight
  return newBox
```

Como exercício, re-escreva o método 'moveRect' para ele criar e retornar um novo Rectangle ao invés de apenas modificar o antigo.

14.9 12.9 Glossário

classe (*class*) Um tipo composto (XXX compound type) definido pelo usuário. Uma classe também pode ser visualizada como um molde que define a forma dos objetos que serão suas instâncias.

instanciar (*instantiate*) Criar uma instância de uma classe.

instância (*instance*) Um objeto que pertence a uma classe.

objeto (*object*) Um tipo de dado composto comumente utilizado para representar uma coisa ou um conceito do mundo real.

construtor (*constructor*) Um método utilizado para criar novos objetos.

atributo (*attribute*) Um dos itens de dados nomeados que compõem uma instância.

igualdade rasa (*shallow equality*) Igualdade de referências; ocorre quando duas referências apontam para o mesmo objeto.

igualdade profunda (*deep equality*) Igualdade de valores; ocorre quando duas referências apontam para objetos que têm o mesmo valor.

cópia rasa (*shallow copy*) Ato de copiar o conteúdo de um objeto, incluindo as referências a objetos embutidos (XXX embedded); implementada pela função `copy` do módulo `copy`.

cópia profunda (*deep copy*) Ato de copiar o conteúdo de um objeto, bem como dos objetos embutidos (XXX embedded), e dos objetos embutidos nestes, e assim por diante; implementada pela função `deepcopy` do módulo `copy`.

CAPÍTULO 15

Capítulo 13: Classes e funções

Tópicos

15.1 13.1 Horario

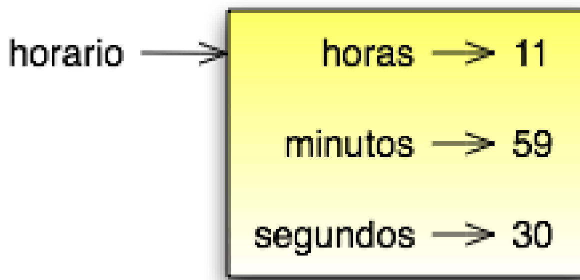
Como exemplo de outro tipo definido pelo usuário, vamos definir uma classe chamada `Horario` que grava os registros de horário do dia. Eis a definição da classe:

```
class Horario:
        pass
```

Podemos criar uma nova instância de `Horario` e determinar atributos para horas, minutos e segundos:

```
horario = Horario()
horario.horas = 11
horario.minutos = 59
horario.segundos = 30
```

O diagrama de estado para o objeto `Horario` parece com isso:



Como exercício, escreva uma função 'imprimirHorario' que tenha como argumento um objeto Horario e imprima-o na forma horas:minutos:segundos.

Como um segundo exercício, escreva uma função booleana que tenha como argumento dois objetos Horario, h1 e h2, e retorne verdadeiro (1) se h1 vem depois de h2 cronologicamente, do contrário, retorne falso (0).

15.2 13.2 Funções Puras

Nas próximas sessões, vamos escrever duas versões de uma função chamada `adicionaHorario`, que calcula a soma de dois horários. Elas vão demonstrar 2 tipos de funções: funções puras e funções modificadoras.

Segue uma versão tosca de `somaHorario`:

```
def somaHorario(h1, h2):
  soma = Horario()
  soma.horas = h1.horas + h2.horas
  soma.minutos = h1.minutos + h2.minutos
  soma.segundos = h1.segundos + h2.segundos
  return soma
```

A função cria um novo objeto `Horario`, inicializa os seus atributos, e retorna uma referência para o novo objeto. Isso é chamado de função pura pois ela não modifica nenhum dos objetos que são passados como parâmetros e não tem nenhum efeito colateral, como imprimir um valor ou pegar entrada do usuário.

Aqui está um exemplo de como usar esta função. Nós vamos criar dois objetos `Horario`: `horarioAtual`, que contém o horário atual; e `horarioDoPao`, que contém a quantidade de tempo que a máquina de fazer pão gasta para fazer pão. Então vamos usar `somaHorario` para tentar saber quando o pão estará pronto. Se você não tiver terminado de escrever `imprimirHorario` ainda, de uma olhada na seção 14.2 antes de você continuar isso:

```
>>> horarioAtual = Horario()
>>> horarioAtual.horas = 9
>>> horarioAtual.minutos = 14
>>> horarioAtual.segundos =  30

>>> horarioDoPao = Horario()
>>> horarioDoPao.horas =  3
>>> horarioDoPao.minutos =  35
>>> horarioDoPao.segundos =  0

>>> horarioTermino = somaHorario(horarioAtual, horarioDoPao)
>>> imprimirHorario(horarioTermino)
```

A saída deste programa é 12:49:30, o que é correto. Por outro lado, existem casos onde o resultado não é correto. Você pode pensar em algum ?

O problema é que esta função não lida com casos onde o número de segundos ou minutos é acrescentado em mais de sessenta. Quando isso acontece, temos de "transportar" os segundos extras para a coluna dos minutos ou os minutos extras na coluna das horas.

Aqui está a segunda versão corrigida da função:

```
def somaHorario(t1, t2):
        soma = Horario()
        soma.horas = t1.horas + t2.horas
        soma.minutos = t1.minutos + t2.minutos
        soma.segundos = t1.segundos + t2.segundos

        if soma.segundos >= 60:
                soma.segundos = soma.segundos - 60
                soma.minutos = soma.minutos + 1

        if soma.minutos >= 60:
                soma.minutos = soma.minutos - 60
                soma.horas = soma.horas + 1

        return soma
```

Apesar desta função estar correta, ela está começando a ficar grande. Depois vamos sugerir uma aproximação alternativa que rende um código menor. Clique aqui para feedback

15.3 13.3 Modificadores

Existem momentos quando é útil para uma função modificar um ou mais dos objetos que ela recebe como parâmetro. Usualmente, quem está chamando a função mantém uma referência para os objetos que ele passa, de forma que quaisquer mudanças que a função faz são visíveis para quem está chamando. Funções que trabalham desta forma são chamadas modificadores.

`incrementar`, que adiciona um número dado de segundos para um objeto Horario, que poderia ser escrito quase naturalmente como um modificador. Um rascunho tosco da função seria algo parecido com isso:

```
def incrementar(horario, segundos):
        horario.segundos = horario.segundos + segundos

        if horario.segundos >= 60:
                horario.segundos = horario.segundos - 60
                horario.minutos = horario.minutos + 1

        if horario.minutos >= 60:
                horario.minutos = horario.minutos - 60
                horario.horas = horario.horas + 1
```

A primeira linha executa a operação básica; o resto lida com os caso especiais que vimos antes.

Esta função esta correta ? O que aconteceria se o parametro segundos for muito maior que sessenta ? Nesse caso, não é suficiente transportar apenas uma vez; teríamos de continuar fazendo isso até que segundos seja menor que sessenta. Uma solução seria substituir os comando if por comandos while:

```
def incrementar(horario, segundos):
        horario.segundos = horario.segundos + segundos

        while horario.segundos >= 60:
                horario.segundos = horario.segundos - 60
                horario.minutos = horario.minutos + 1
```

```
    while horario.minutos >= 60:
        horario.minutos = horario.minutos - 60
        horario.horas = horario.horas + 1
```

Esta função agora esta correta, mas não é a solução mais eficiente.

Como um exercício, reescreva esta função de maneira que ela não contenha nenhum loop. Como um segundo exercício, reescreva `incrementar` como uma função pura, e escreva chamadas de funções para as duas funções. Clique aqui para feedback

15.4 13.4 O que é melhor ?

Qualquer coisa que pode ser feita com modificadores também podem ser feitas com funções puras. De fato, algumas linguagens de programação permitem apenas funções puras. Existe alguma evidência que programas que usam funções puras são desenvolvidos mais rapidamente e são menos propensos a erros que programas que usam modificadores. No entanto, modificadores as vezes são convenientes, e em alguns casos, programação funcional é menos eficiente.

Em geral, recomendamos que você escreva funções puras sempre que for necessário e recorrer para modificadores somente se existir uma grande vantagem. Esta aproximação poderia ser chamada de um estilo de programação funcional. Clique aqui para feedback

15.5 13.5 Desenvolvimento Prototipado versus Desenvolvimento Planejamento

Neste capítulo, demonstramos uma aproximação para o desenvolvimento de programas que chamamos de desenvolvimento prototipado. Em cada caso, escrevemos um rascunho tosco (ou prototipo) que executou os cálculos básicos e então, o testamos em uns poucos casos, corrigindo então, as falhas que fomos encontrando.

Embora esta aproximação possa ser eficaz, ela pode conduzir para código que é desnecessariamente complicado desde que trata de muitos casos especiais e unreliable desde que é difícil saber se você encontrou todos os erros.

An alternative is planned development, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60! The second component is the "ones column," the minute component is the "sixties column," and the hourcomponent is the "thirty-six hundreds column."

When we wrote addTime and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem we can convert a Time object into a single number and take advantage of the fact that the computer knows how to do arithmetic with numbers. The following function converts a Timeobject into an integer:

```
def converterParaSegundos(t):
  minutos = t.horas * 60 + t.minutos
  segundos = minutos * 60 + t.segundos
  return segundos
```

Agora, tudo que precisamos é uma maneira de converter de um inteiro para um objeto `Horario`:

```
def criarHorario(segundos):
    horario = Time()
    horario.horas = segundos/3600
    segundos = segundos - horario.horas * 3600
```

```
    horario.minutos = segundos/60
    segundos = segundos - horario.minutos * 60
    horario.segundos = segundos
    return horario
```

Você deve ter que pensar um pouco para se convencer que esta técnica de converter de uma base para outra é correta. Assumindo que você está convencido, você pode usar essas funções para reescrever somaHorario:

```
def somaHorario(t1, t2):
    segundos = converterParaSegundos(t1) + converterParaSegundos(t2)
    return criarHorario(segundos)
```

Esta versão é muito mais curta que a original, e é muito mais fácil para demonstrar que está correta (assumindo, como sempre, que as funções que são chamadas estão corretas).

Como um exercício, reescreva `incrementar` da mesma forma. Clique aqui para feedback

15.6 13.6 Generalização

Algumas vezes, converter de base 60 para base 10 e voltar é mais difícil do que simplesmente lidar com horários. Conversão de base é mais abstrata; nossa intuição para lidar com horários é melhor.

Mas se XXXX But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (converterParaSeguntos e criarHorario), nós conseguimos um programa que é menor, fácil de ler e depurar, e mais confiável.

É também fácil para adicionar funcionalidades depois. Por exemplo, imagine subtrair dois Horarios para encontrar a duração entre eles. Uma aproximação ingênua seria implementar subtração com empréstimo (?? borrowing - Isso mesmo ??). Usando as funções de conversão será mais fácil e provavelmente estará correto.

Ironicamente, algumas vezes fazer um problema mais difícil (ou mais genérico) o torna mais simples (porque existem alguns poucos casos especiais e poucas oportunidades para errar). Clique aqui para feedback

15.7 13.7 Algoritmos

Quando você escreve uma solução genérica para uma classe de problemas, ao contrário de uma solução específica para um único problema, você escreveu um algorítmo. Nós mencionamos isso antes mas não definimos cuidadosamente. Isso não é fácil para definir, então nós vamos tentar //a couple of approaches//.

Primeiramente, considere alguma coisa que não seja um algorítmo. Quando você aprendeu a multiplicar números de um dígito, você provavelmente memorizou a tabela de multiplicação. Como resultado, você memorizou 100 soluções específicas. Esse tipo de conhecimento não é algorítmo.

Mas se você é "preguiçoso", você provavelmente trapaceou por ter aprendido alguns truques. Por exemplo, para encontrar o produto de n e 9, você pode escrever n-1 como o primeiro dígito e 10-n como o segundo dígito. Este truque é um solução genérica para multiplicar qualquer número de um dígito por 9. Isso é um algoritmo!

De modo parecido, as técnicas que você aprendeu para adicionar //com transporte//, //subtraction with borrowing//, e divisão longa são todas algorítmos. Uma das características dos algorítmos é que eles não requerem nenhuma inteligência para serem executados (carry out). Eles são processos mecânicos no qual cada passo segue o último de acordo com um conjunto simples de regras.

Na nossa opinião, é preocupante que humanos gastem tanto tempo na escola aprendendo a executar algoritmos que, literalmente, não requerem inteligência.

Por outro lado, o processo de projetar algoritmos é interessante, intelectualmente desafiante, e uma parte central daquilo que chamamos programação.

Algumas das coisas que as pessoas fazem naturalmente, sem dificuldade ou consciência, são as mais difíceis de se expressar através de algoritmos. Entender a linguagem natural é um bom exemplo. Todos nós fazemos isso, mas até hoje ninguém conseguiu explicar como fazemos isso, pelo menos não na forma de algoritmo. Clique aqui para feedback.

15.8 13.8 Glossário

função pura (*pure function*) Uma função que não modifica nenhum dos objetos que ela recebe como parâmetro. A maioria das funções puras é frutífera.

modificador (*modifier*) Uma função que muda um ou mais dos objetos que ela recebe como parâmetros. A maioria dos modificadores é nula.

estilo de programação funcional (*functional programming style*) Um estilo de programação onde a maioria das funções são puras.

desenvolvimento prototipado (*prototype development*) Uma maneira de desenvolver programas começando com um protótipo e gradualmente melhorando-o.

desenvolvimento planejado (*planned development*) Uma maneira de desenvolver programas que envolvem uma percepção de alto nível do problema e mais planejamento do que desenvolvimento incremental ou desenvolvimento prototipado.

algoritmo (*algorithm*) Um conjunto de instruções para resolver uma classe de problemas usando um processo mecânico, não inteligente.

CAPÍTULO 16

Capítulo 14: Classes e métodos

Tópicos

ATENÇÃO As referências cruzadas a nomes em códigos de outros capítulos (especialmente 13) ainda não foram unificadas...

16.1 14.1 Características da orientação a objetos

Python é uma **linguagem de programação orientada a objetos**, o que significa que ela tem características que suportam a **programação orientada a objetos**.

Não é fácil definir programação orientada a objetos, mas temos visto already algumas de suas características:

- Programas são construídos sobre definições de objetos e definições de funções, e a maioria das computações é expressa em termos de operações sobre objetos.
- Cada definição de objeto corresponde a algum objeto ou conceito do mundo real, e as funções que operam com aqueles objetos correspondem à maneira como os objetos do mundo real interagem.

Por exemplo, a classe `Tempo`, definida no capítulo 13 corresponde à maneira como as pessoas registram as horas do dia, e as funções que definimos correspondem aos tipos de coisas que as pessoas fazem com times. Do mesmo modo, as classes `Ponto` e `Retângulo` correspondem aos conceitos matemáticos de um ponto e de um retângulo.

Até aqui, não tiramos vantagem das características fornecidas por Python que suportam a programação orientada a objetos. Estritamente falando, estas características não são necessárias. Na maior parte das vezes, elas fornecem uma sintaxe alternativa para as coisas que já fizemos, mas em muitos casos, a alternativa é mais concisa e convém mais acuradamente à estrutura do programa.

Por exemplo, no programa `Time`, não existe uma conexão óbvia entre a definição da classe e a definição da função que segue. Com alguma investigação, fica aparente que toda função toma pelo menos um objeto `Time` como um parâmetro.

Esta observação é a motivação por trás dos **métodos**. Já temos visto alguns métodos, tais como `keys` (`chaves`) e `values` (`valores`), os quais foram invocados em dicionários. Cada método é associado com uma classe e é intended para ser invocado em instâncias daquela classe.

Métodos são simplesmente como funções, com duas diferenças:

- Métodos são definidos dentro da definição de uma classe para tornar explícita a relação entre a classe e o método.
- A sintaxe para a chamada do método é diferente da sintaxe para a chamada de uma função.

Nas próximas seções, vamos pegar as funções dos dois capítulos anteriores e transformá-las em métodos. Esta transformação é puramente mecânica: você pode conseguí-la simplesmente seguindo uma seqüência de passos. Se você se sentir confortável convertendo de uma forma para a outra, você estará apto para escolher a melhor forma para seja o lá o que for que você estiver fazendo.

16.2 14.2 exibeHora (printTime)

No capítulo 13, definimos uma classe chamada `Horário` (`Time`) e você escreveu uma função chamada `exibeHora` (`printTime`), que deve ter ficado mais ou menos assim:

```
class Horario:
  pass

def exibeHora(time)
  print str(time.horas) + ?:? + \
    str(time.minutos) + ?:? + \
    str(time.segundos)
```

Para chamar esta função, passamos um objeto `Time` como um parâmetro:

```
>>> horaCorrente = Hora()
>>> horaCorrente.horas = 9
>>> horaCorrente.minutos = 14
>>> horaCorrente.segundos = 30
>>> exibeHora(horaCorrente)
```

Para fazer de `exibeHora` um método, tudo o que temos a fazer é mover a definição da função para dentro da definição da classe. Note a mudança na endentação:

```
class Horario:
  def exibeHora(time):
    print str(time.horas) + ?:? + \
      str(time.minutos) + ?:? + \
      str(time.segundos)
```

Agora podemos chamar `exibeHora` usando a natação de ponto:

```
>>> horaCorrente.exibeHora()
```

Como é usual, o objeto no qual o método é invocado aparece antes do ponto e o nome do método aparece depois do ponto.

O objeto no qual o método é invocado é atribuído ao primeiro parâmetro, então, neste caso, `horaCorrente` é atribuído ao parâmetro `time`.

Por convenção, o primeiro parâmetro de um método é chamado `self`. A razão para isto é um pouco convoluted, mas é baseada numa metáfora útil.

A sintaxe para uma chamada de função, `exibeHora(horaCorrente)`, sugere que a função é um agente ativo. Diz algo como, ?Ei, `exibeHora`! Aqui está um objeto para você exibir.?

Na programação orientada a objetos, os objetos são agentes ativos. Uma chamado do tipo `horaCorrente.exibeHora()` diz ?Ei, `horaCorrente`! Por favor exiba-se a si mesmo!?

Esta mudança de perspectiva pode ser mais polida, mas não fica óbvio que seja útil. Nos exemplos que temos visto até aqui, pode ser que não seja. Mas às vezes, deslocar a responsabilidade das funções para cima dos objetos torna possível escrever funções mais versáteis, e torna mais fácil manter e reutilizar o código.

16.3 14.3 Um outro exemplo

Vamos converter `incremento` (da Seção 13.3) em um método. Para poupar espaço, deixaremos de fora métodos definidos previamente(anteriormente?), mas você deve mantê-los em sua versão:

```
class Time:
  #previous method definitions here...

  def increment(self, segundos):
  self.seconds = seconds + self.seconds

  while self.segundos >= 60:
    self.seconds = self.segundos - 60
    self.minutes = self.minutos + 1

  while self.minutes >= 60:
    self.minutes = self.minutos - 60
    self.hours = self.horas + 1
```

A transformação é puramente mecânica ? movemos a definição do método para dentro da definição da classe e mudamos o nome do primeiro parâmetro.

Agora podemos chamar `incremento` como um método:

```
horaCorrente.incremento(500)
```

De novo, o objeto no qual o método é chamado gets atribui ao primeiro parâmetro, `self`. O segundo parâmetro, `segundo` toma(gets) o valor `500`.

> *Como um exercício, converta ?converteParaSegundos? (da Seção 13.5) para um método na classe ?Time?.*

16.4 14.4 Um exemplo mais complicado

...

16.5 14.10 Glossário

linguagem orientada a objetos Uma linguagem que provê características tais como classes definidas pelo usuário e herança, que facilitam a programação orientada a objetos.

programação orientada a objetos Um estilo de programação na qual os dados e as operações que os manipulam estão organizados em classes e métodos.

método Uma função que é definida dentro de uma definição de classe e é chamada em instâncias desta classe.

override (sem traducao; termo consagrado) Substituir uma definição já pronta. Exemplos incluem substituir um parâmetro padrão por um argumento particular e substituir um método padrão, fornecendo um novo método com o mesmo nome.

método de inicialização (tambem chamado de construtor) Um método especial que é invocado automaticamente quando um novo objeto é criado e que inicializa os atributos deste objeto.

sobrecarga de operador Estender a funcionalidade dos operadores nativos (+, -, *, >, <, etc.) de forma que eles funcionem também com tipos definidos pelo usuário.

produto escalar Operação definida na álgebra linear que multiplica dois pontos (com coordenadas (x,y,z)) e retorna um valor numérico.

multiplicação por escalar Operação definida na álgebra linear que multiplica cada uma das coordenadas de um ponto por um valor numérico.

polimórfica Uma função que pode operar com mais de um tipo. Se todas as operações de uma função pode ser aplicadas a um certo tipo, então a função pode ser aplicada a este tipo.

CAPÍTULO 17

Capítulo 15: Conjuntos de objetos

Tópicos

17.1 15.1 Composição

Até agora, você vio diversos exemplos de composição. Um dos primeiros exemplos foi o uso de uma invocação de método como parte de uma expressão. Outro exemplo é a estrutura aninhada dos comandos: você pode pôr um comando `if` dentro de um laço `while`, dentro de outro comando `if`, e assim por diante.

Tendo visto este padrão, e tendo aprendido a respeito de listas e objetos, você não deveria ficar surpreso em aprender que você pode criar listas de objetos. Você também pode criar obejtos que contêm listas (como atritos); você pode criar listas que contêm listas; você pode criar objetos que contêm objetos; e assim por diante.

Neste capítulo e no próximo, você irá ver alguns exemplos destas combinações, usando objetos `Carta` como exemplo.

17.2 15.2 Objetos `Carta`

Se você não estiver familiarizado com jogos de cartas, agora é um bom momento para conseguir um baralho, ou então esse capítulo pode não fazer muito sentido. Há 52 cartas em um baralho, cada uma das quais pertence a um dos quatro naipes e a uma das treze posições. Os naipes são Espadas, Copas, Ouros e Paus (em ordem descendente no *bridge*). As posições são Ás, 2, 3, 4, 5, 6, 7, 8, 9, 10, Valete, Rainha e Rei. Dependendo do jogo, a posição do Ás pode ser maior do que a do Rei ou menor do que a do 2.

Se quisermos definir um novo objeto para representar uma carta, é óbvio que os atributos devem ser `posicao` e `naipe`. Não tão óbvio são os tipos aos quais devem pertencer os atributos. Uma possibilidade é usar strings contendo palavras como "`Espada`" para naipes e "`Rainha`" para posições. Um problema com esta implementação é que não seria fácil comparar cartas para ver qual possui o maior naipe ou posição.

Uma alternativa é usar inteiros para **codificar** as posições e naipes. "Codificar", neste caso, não significa o mesmo que as pessoas normalmente pensam, que é criptografar ou traduzir para um código secreto. O que um cientista da computação quer dizer com "codificar" é "definir um mapeamento entre uma seqüência de números e os itens que eu quero representar". Por exemplo:

- Espadas -> 3
- Copas -> 2
- Ouros -> 1
- Paus -> 0

Uma característica óbvia deste mapeamento é que os naipes são mapeados para inteiros na ordem, de modo que nós podemos comparar naipes pela comparação de inteiros. O mapeamento de posições é bastante óbvio. Cada uma das posições numéricas mapeia para o inteiro correspondente e, as cartas com figura são mapeadas conforme abaixo:

- Valete -> 11
- Rainha -> 12
- Rei -> 13

O motivo pelo qual nós estamos usando notação matemática para estes mapeamentos é que eles não são parte do programa Python. Eles são parte do projeto do programa, mas eles nunca aparecem explicitamente no código. A definição de classe para o tipo `Carta` fica parecida com esta:

```
class Carta:
  def __init__(self, naipe=0, posicao=0):
    self.naipe = naipe
    self.posicao = posicao
```

Como sempre, nós fornecemos um método de inicialização que recebe um parâmetro opcional para cada atributo.

Para criar um objeto que representa o 3 de Paus, usa-se este comando:

```
tresDePaus = Carta(0, 3)
```

O primeiro argumento, 0, representa o naipe de Paus.

17.3 15.3 Atributos de classe e o método `__str__`

Para imprimir objetos `Carta` de uma maneira que as pessoas possam facilmente ler, nós gostaríamos de mapear os códigos inteiros para palavras. Uma forma natural de fazer isso é usar listas de strings. Nós atribuímos estas listas para **atributos de classe** no topo da definição de classe:

```
class Carta:
  listaDeNaipes = ["Paus", "Ouros", "Copas", "Espadas"]
  listaDePosicoes = ["narf", "Ás", "2", "3", "4", "5", "6", "7",
                     "8", "9", "10", "Valete", "Rainha", "Rei"]

  # método init omitido

  def __str__(self):
```

```
    return (self.listaDePosicoes[self.posicao] + " de " +
            self.ListaDeNaipes[self.naipe])
```

Um atributo de classe é definido fora de qualquer método, e ele pode ser acessado por quaisquer métodos da classe.

Dentro de `__str__`, nós podemos usar `listaDeNaipes` e `listaDePosicoes` para mapear os valores numéricos de `naipe` e `posicao` para strings. Por exemplo, a expressão `self.listaDeNaipes[self.naipe]` significa "use o atributo `naipe` do objeto `self` como um índice para o atributo de classe chamado `listaDeNaipes`, e selecione a string apropriada".

O motivo para o "`narf`" no primeiro elemento em `listaDePosicoes` é preencher o lugar do 0-ésimo elemento da lista, que nunca será usado. As únicas posições válidas são de 1 a 13. Este item desperdiçado não é inteiramente necessário. Nós poderíamos ter iniciado com 0, como é normal. Porém, é menos confuso codificar 2 como 2, 3 como 3, e assim por diante.

Com os métodos que nós temos até agora, nós podemos criar e imprimir cartas:

```
>>> carta1 = Carta(1, 11)
>>> print carta1
Valete de Ouros
```

Atributos de classe como `listaDeNaipes` são compartilhados por todos os objetos `Carta`. A vantagem disso é que nós podemos usar qualquer objeto `Carta` para acessar os atributos de classe:

```
>>> carta2 = Carta(1, 3)
>>> print carta2
3 de Ouros
>>> print carta2.listaDeNaipes[1]
Ouros
```

A desvantagem é que se nós modificarmos um atributo de classe, isso afetará cada instância da classe. Por exemplo, se nós decidirmos que "Valete de Ouros" deveria realmente se chamar "Valete de Baleias Rodopiantes", nós poderíamos fazer isso:

```
>>> carta1.listaDeNaipes[1] = "Baleias Rodopiantes"
>>> print carta1
Valete de Baleias Rodopiantes
```

O problema é que *todos* os Ouros se tornam Baleias Rodopiantes:

```
>>> print carta2
Valete de Baleias Rodopiantes
```

Normalmente, não é uma boa idéia modificar atributos de classe.

17.4 15.4 Comparando cartas

Para tipos primitivos, existem operadores condicionais (<, >, ==, etc.) que comparam valores e determinam quando um é maior que, menor que ou igual a outro. Para tipos definidos pelo usuário, nós podemos sobrescrever o comportamento dos operadores pré-definidos fornecendo um método `__cmp__`. Por convenção, `__cmp__` recebe dois parâmetros, `self` e `other`, e retorna 1 se o primeiro objeto for maior, -1 se o segundo objeto for maior, e 0 se eles forem iguais.

Alguns tipos são totalmente ordenados, o que significa que nós podemos comparar quaisquer dois elementos e dizer qual é o maior. Por exemplo, os inteiros e os números de ponto flutuante são totalmente ordenados. Alguns conjuntos são não-ordenados, o que significa que não existe maneira significativa de dizer que um elemento é maior que o outro. Por exemplo, as frutas são não-ordenadas, e é por isso que não podemos comparar maçãs e laranjas.

O conjunto de cartas de jogo é parcialmente ordenado, o que significa que às vezes você pode comparar cartas, e às vezes não. Por exemplo, você sabe que o 3 de Paus é maior do que o 2 de Paus, e que o 3 de Ouros é maior do que o 3 de Paus. Mas qual é o melhor, o 3 de Paus ou o 2 de Ouros? Um tem uma posição maior, mas o outro tem um naipe maior.

Para tornar as cartas comparáveis, você tem que decidir o que é mais importante: posição ou naipe. Para ser honesto, a escolha é arbitrária. Por questão de escolha, nós iremos dizer que naipe é mais importante, porque um baralho de cartas novo vem ordenado com todas as cartas de Paus juntas, seguidas pelas de Ouros, e assim por diante.

Com essa decisão, nós podemos escrever `__cmp__`:

```
def __cmp__(self, other):
  # verificar os naipes
  if self.naipe > other.naipe: return 1
  if self.naipe < other.naipe: return -1
  # as cartas têm o mesmo naipe... verificar as posições
  if self.posicao > other.posicao: return 1
  if self.posicao < other.posicao> return -1
  # as posições são iguais... é um empate
  return 0
```

Nesta ordenação, Ases são menores do que 2.

> *Como um exercício, modifique ‘‘__cmp__‘‘, de modo que os Ases sejam maiores do que os Reis.*

17.5 15.5 Baralhos

Agora que nós temos objetos para representar `Cartas`, o próximo passo lógico é definir uma classe para representar um `Baralho`. É claro que um baralho é formado por cartas; portanto, cada objeto `Baralho` irá conter uma lista de cartas como um atributo.

A seguir, damos uma definição para a classe `Baralho`. O método de inicialização cria o atributo `cartas` e gera o conjunto padrão de 52 cartas:

```
classe Baralho
  def __init__(self):
    self.cartas = []
    for naipe in range(4):
      for posicao in range(1, 14):
        self.cartas.append(Carta(naipe, posicao))
```

A maneira mais fácil de popular o baralho é com um laço aninhado. O laço externo enumera os naipes de 0 até 3. O laço interno enumera as posições de 1 até 13. Como o laço externo repete quatro vezes e o laço interno 13 vezes, o número total de vezes que o corpo é executado é 52 (13 vezes quatro). Cada iteração cria uma nova instância de `Carta` com o naipe e posição atuais e a inclui na lista `cartas`.

O método `append` trabalha sobre listas mas não, obviamente, sobre tuplas.

17.6 15.6 Imprimindo o baralho

Como sempre, quando nós definimos um novo tipo de objeto, nós gostaríamos de ter um método para imprimir o conteúdo de um objeto. Para imprimir um `Baralho`, nós percorremos a lista e imprimimos cada `Carta`:

```
class Baralho:
  ...
  def imprimirBaralho(self):
```

```
    for carta in self.cartas:
      print carta
```

Aqui, e a partir daqui, as reticências (...) indicam que nós omitimos os outros métodos da classe.

Como uma alternativa a `imprimirBaralho`, nós poderíamos escrever um método `__str__` para a classe `Baralho`. A vantagem de `__str__` é que ela é mais flexível. Em vez de apenas imprimir o conteúdo de um objeto, ela gera uma representação em string que outras partes do programa podem manipular antes de imprimir ou armazenar para uso posterior.

Abaixo, uma versão de `__str__` que devolve uma representação em string de um `Baralho`. Para adicionar um pouco de estilo, ela distribui as cartas em uma cascata, na qual cada carta é indentada um espaço a mais do que a carta anterior:

```
class Baralho:
  ...
  def __str__(self):
    s = ""
    for i in range(len(self.cartas)):
      s = s + " "*i + str(self.cartas[i]) + "\n"
    return s
```

Este exemplo demonstra diversas características. Primeiro, em vez de percorrer `self.cartas` e atribuir cada carta a uma variável, nós estamos usando `i` como uma variável de laço e um índice para a lista de cartas.

Segundo, nós estamos usando o operador de multiplicação de strings para indentar cada carta com um espaço adicional com relação à anterior. A expressão `" "*i` produz um número de espaços igual ao valor atual de `i`.

Terceiro, em vez de usar o comando `print` para imprimir as cartas, nós usamos a função `str`. Passar um objeto como um argumento para `str` equivale a invocar o método `__str__` sobre o objeto.

Finalmente, nós estamos usando a variável `s` como um **acumulador**. Inicialmente, `s` é a string vazia. A cada repetição do laço, uma nova string é gerada e concatenada com o valor antigo de `s` para obter um novo valor. Quando o laço termina, `s` contém a representação em string completa do `Baralho`, que se parece com:

```
>>> baralho = Baralho()
>>> print Baralho
Ás de Paus
 2 de Paus
  3 de Paus
   4 de Paus
    5 de Paus
     6 de Paus
      7 de Paus
       8 de Paus
        9 de Paus
         10 de Paus
          Valete de Paus
           Rainha de Paus
            Rei de Paus
             Ás de Ouros
```

E assim por diante. Mesmo que o resultado apareça em 52 linhas, é uma string longa que contém *newlines*.

17.7 15.7 Embaralhando

Se um baralho estiver perfeitamente embaralhado, então cada carta tem a mesma probabilidade de aparecer em qualquer lugar no baralho, e qualquer localização no baralho tem a mesma probabilidade de conter qualquer carta.

Para embaralhar as cartas, nós usaremos a função `randrange` do módulo `random`. Com dois argumentos inteiros, `a` e `b`, `randrange` escolhe um inteiro aleatório no intervalo `a <= x < b`. Como o limite superior é estritamente menor que `b`, nós podemos usar o comprimento de uma lista como o segundo parâmetro, e nós garantimos que o índice sempre será válido. Por exemplo, esta expressão escolhe o índice de uma carta aleatória em um baralho:

```
random.randrange(0, len(self.cartas))
```

Uma maneira fácil de embaralhar as cartas é percorrer a lista e trocar cada carta por outra escolhida aleatoriamente. É possível que a carta seja trocada por ela mesma, mas isso não é problema. Na verdade, se nós excluíssemos essa possibilidade, a ordem das cartas não seria totalmente aleatória:

```
class Baralho:
  ...
  def embaralhar(self):
    import random
    nCartas = len(self.cartas)
    for i in range(nCartas):
      j = random.randrange(i, nCartas)
      self.cartas[i], self.cartas[j] = self.cartas[j], self.cartas[i]
```

Em vez de assumir que existem 52 cartas no baralho, nós obtivemos o comprimento real da lista e o guardamos na variável `nCartas`.

Para cada carta no baralho, nós escolhemos uma carta aleatória dentre as cartas que ainda não foram embaralhadas. Então, nós trocamos a carta atual (`i`) pela carta selecionada (`j`). Para trocar as cartas, nós usamos uma atribuição de tupla, como visto na Seção 9.2:

```
self.cartas[i], self.cartas[j] = self.cartas[j], self.cartas[i]
```

Como exercício, reescreva esta linha de código sem usar uma atribuição de seqüência.

17.8 15.8 Removendo e distribuindo cartas

Outro método que pode ser útil para a classe `Baralho` é `removerCarta`. Ele recebe uma carta como parâmetro, remove-a do baralho e retorna verdadeiro (1), se a carta estava no baralho e falso (0), caso contrário:

```
class Baralho:
  ...
  def removerCarta(self, carta):
    if carta in self.cartas:
      self.cartas.remove(carta)
      return 1
    else
      return 0
```

O operador `in` retorna verdadeiro se o primeiro operando estiver contido no segundo, que deve ser uma lista ou uma tupla. Se o primeiro operando for um objeto, Python usa o método __cmp__ do objeto para determinar igualdade com os itens da lista. Como o método __cmp__ da classe `Carta` verifica por igualdade profunda, o método `removerCarta` também testa por igualdade profunda.

Para distribuir as cartas, nós iremos remover e devolver a carta do topo. O método de lista `pop` fornece uma maneira conveniente de fazer isso:

```
class Baralho:
  ...
  def distribuirCarta(self):
    return self.cards.pop()
```

Na verdade, `pop` remove a *última* carta da lista. Portanto, nós estamos realmente distribuindo as cartas do fim para o início do baralho.

Uma última operação que nós poderíamos querer é a função booleana `estahVazio`, que retorna verdadeiro se o baralho não contém cartas:

```
class Baralho:
  ...
  def estahVazio(self):
    return (len(self.cartas) == 0)
```

17.9 15.9 Glossário

codificar (*encode*) Representar um conjunto de valores usando outro conjunto de valores, construindo um mapeamento entre eles.

atributo de classe (*class atribute*) Uma variável que é definida dentro de uma definição de classe, mas fora de qualquer método. Atributos de classe podem ser acessados a partir de qualquer método da classe e são compartilhados por todas as instâncias da classe.

acumulador (*accumulator*) Uma variável usada em um laço para acumular uma série de valores, para, por exemplo, concatená-los em uma string ou somá-los a uma soma em andamento.

Capitulo 16: Herança

Tópicos

18.1 16.1 Herança

Uma das características mais marcantes das linguagens orientadas a objetos é a **herança**. Herança é a habilidade de definir uma nova classe que é uma versão modificada de uma classe existente.

A principal vantagem dessa característica é que você pode adicionar novos métodos a uma classe sem ter que modificar a classe existente. Chama-se "herança" porque a nova classe herda todos os métodos da classe existente. Ampliando a metáfora, podemos dizer que a classe existente é às vezes chamada de classe **mãe** (*parent*). A nova classe pode ser chamada de classe **filha** ou, simplesmente, "subclasse".

A herança é uma característica poderosa. Alguns programas que seriam complicados sem herança podem ser escritos de forma simples e concisa graças a ela. E a herança também pode facilitar o reuso do código, uma vez que você pode adaptar o comportamento de classes existentes sem ter que modificá-las. Em alguns casos, a estrutura da herança reflete a natureza real do problema, tornando o programa mais fácil de entender.

Por outro lado, a herança pode tornar um programa seja difícil de ler. Quando um método é invocado, às vezes não está claro onde procurar sua definição. A parte relevante do código pode ser espalhada em vários módulos. E, também, muitas das coisas que podem ser feitas utilizando herança também podem ser feitas de forma igualmente elegante (ou até mais) sem ela. Se a estrutura natural do problema não se presta a utilizar herança, esse estilo de programação pode trazer mais problemas que vantagens.

Nesse capítulo, vamos demonstrar o uso de herança como parte de um programa que joga uma variante de Mico. Um dos nossos objetivos é escrever um código que possa ser reutilizado para implementar outros jogos de cartas.

18.2 16.2 Uma mão de cartas

Para quase todos os jogos de baralho, é preciso representar uma mão de cartas. Uma mão de cartas é similar a um maço de baralho. Porque ambos são formados por uma série de cartas e ambos requerem operações, como, adicionar e remover cartas. Fora isso, a habilidade de embaralhar a mão e o baralho também são úteis.

Mas, ao mesmo tempo, a mão é também diferente do baralho. Dependendo do jogo que está sendo jogado, precisamos realizar algumas operações nas mãos de cartas que não fazem sentido para o baralho inteiro. Por exemplo, no pôquer, podemos classificar uma mão (trinca, flush, etc.) ou compará-la com outra mão. No jogo de bridge, podemos querer computar a quantidade de pontos que há numa mão, a fim de fazer um lance.

Essa situação sugere o uso de herança. Se `Mao` é uma subclasse de `Baralho`, terá todos os métodos de `Baralho`, e novos métodos podem ser adicionados.

Na definição de classe, o nome da classe pai aparece entre parênteses:

```
class Mao(Baralho):
  pass
```

Esse comando indica que a nova classe `Mao` herda da classe existente `Baralho`.

O construtor de `Mao` inicializa os atributos da mão, que são `nome` e `cartas`. A string `nome` identifica essa mão, provavelmente pelo nome do jogador que está segurando as cartas. O nome é um parâmetro opcional com a string vazia como valor default. `cartas` é a lista de cartas da mão, inicializada com uma lista vazia

```
class Mao(Baralho):
  def __init__(self, nome=""):
    self.cartas = []
    self.nome = nome
```

Em praticamente todos os jogos de cartas, é necessario adicionar e remover cartas do baralho. Remover cartas já está resolvido, uma vez que `Mao` herda `removerCarta` de `Baralho`. Mas precisamos escrever `adicionarCarta`:

```
class Mao(Baralho):
  #...
  def adicionarCarta(self,carta):
    self.cartas.append(carta)
```

De novo, a elipse indica que omitimos outros métodos. O método de listas `append` adiciona a nova carta no final da lista de cartas.

18.3 16.3 Dando as cartas

Agora que temos uma classe `Mao`, queremos distribuir cartas de `Baralho` para mãos de cartas. Não é imediatamente óbvio se esse método deve ir na classe `Mao` ou na classe `Baralho`, mas como ele opera num único baralho e (possivelmente) em várias mãos de cartas, é mais natural colocá-lo em `Baralho`.

O método `distribuir` deve ser bem geral, já que diferentes jogos terão diferentes requerimentos. Podemos querer distribuir o baralho inteiro de uma vez só ou adicionar uma carta a cada mão.

`distribuir` recebe dois argumentos, uma lista (ou tupla) de mãos e o numero total de cartas a serem dadas. Se não houver cartas suficientes no baralho, o método dá todas as cartas e pára:

```
class Baralho:
  #...
  def distribuir(self, maos, nCartas=999):
    nMaos = len(maos)
    for i in range(nCartas):
```

```
    if self.estahVazia(): break    # interromper se acabaram as cartas
    carta = self.pegarCarta()      # pegar a carta do topo
    mao = maos[i % nMaos]       # quem deve receber agora?
    mao.adicionarCarta(carta)   # adicionar a carta à mao
```

O segundo parâmetro, `nCartas`, é opcional; o default é um número grande, o que na prática significa que todas as cartas do baralho serão dadas se este parâmetro for omitido.

A variável do laço `i` vai de 0 a `nCartas-1`. A cada volta do laço, uma carta é removida do baralho, usando o método de lista `pop`, que remove e retorna o último item na lista.

O operador módulo (%) permite dar cartas em ao redor da mesa (uma carta de cada vez para cada mão). Quando `i` é igual ao numero de mãos na lista, a expressão `i % nMaos` volta para o começo da lista (índice 0).

18.4 16.4 Exibindo a mao

Para exibir o conteúdo de uma mão, podemos tirar vantagem dos métodos `exibirBaralho` e `__str__` herdados de `Baralho`. Por exemplo:

```
>>> baralho = Baralho()
>>> baralho.embaralhar()
>>> mao = Mao("fabio")
>>> baralho.distribuir([mao], 5)
>>> print mao
Mão fabio contém
2 de espadas
 3 de espadas
  4 de espadas
   Ás de copas
    9 de paus
```

Nao é lá uma grande mão, mas tem potencial para um *straight flush*.

Embora seja conveniente herdar os métodos existentes, há outras informacoes num objeto `Mao` que podemos querer incluir quando ao exibí-lo. Para fazer isso, podemos fornecer um método `__str__` para a classe `Mao` que sobrescreva o da classe `Baralho`:

```
class Mao(Baralho)
  #...
  def __str__(self):
    s = "Mao " + self.nome
    if self.estahVazia():
      return s + " está vazia\n"
    else:
      return s + " contém\n" + Baralho.__str__(self)
```

Inicialmente, `s` é uma string que identifica a mão. Se a mão estiver vazia, o programa acrescenta as palavras `está vazia` e retorna o resultado.

Se não, o programa acrescenta a palavra `contém` e a representação de string do `Baralho`, computada pela invocação do método `__str__` na classe `Baralho` em `self`.

Pode parecer estranho enviar `self`, que se refere à `Mao` corrente, para um método `Baralho`, mas isso só até voce se lembrar que um `Mao` é um tipo de `Baralho`. Objetos `Mao` podem fazer tudo que os objetos `Baralho` fazem, entao, é permitido passar uma instância de `Mao` para um método `Baralho`.

Em geral, sempre é permitido usar uma instância de uma subclasse no lugar de uma instância de uma classe mãe.

18.5 16.5 A classe `JogoDeCartas`

A classe `JogoDeCartas` toma conta de algumas tarefas básicas comuns a todos os jogos, como, criar o baralho e embaralhá-lo:

```
class JogoDeCartas:
  def __init__(self):
    self.baralho = Baralho()
    self.baralho.embaralhar()
```

Este é o primeiro dos casos que vimos até agora em que o método de inicialização realiza uma computação significativa, para além de inicializar atributos.

Para implementar jogos específicos, podemos herdar de `JogoDeCartas` e adicionar caracteristicas para o novo jogo. Como exemplo, vamos escrever uma simulação de Mico.

O objetivo do jogo é livrar-se das cartas que estiverem na mão. Para fazer isso, é preciso combinar cartas formando pares ou casais que tenham a mesma cor e o mesmo número ou figura. Por exemplo, o 4 de paus casa com o 4 de espadas porque os dois naipes são pretos. O Valete de copas combina com o Valete de ouros porque ambos são vermelhos.

Antes de mais nada, a Dama de paus é removida do baralho, para que a Dama de espadas fique sem par. A Dama de espadas então faz o papel do mico. As 51 cartas que sobram são distribuidas aos jogadores em ao redor da mesa (uma carta de cada vez para cada mão). Depois que as cartas foram dadas, os jogadores devem fazer todos os casais possíveis que tiverem na mão, e em seguida descartá-los na mesa.

Quando ninguém mais tiver nenhum par para descartar, o jogo começa. Na sua vez de jogar, o jogador pega uma carta (sem olhar) do vizinho mais proximo à esquerda, que ainda tiver cartas. Se a carta escolhida casar com uma carta que ele tem na mão, ele descarta esse par. Quando todos os casais possíveis tiverem sido feitos, o jogador que tiver sobrado com a Dama de espadas na mão perde o jogo.

Em nossa simulação computacional do jogo, o computador joga todas as mãos. Infelizmente, algumas nuances do jogo presencial se perdem. Num jogo presencial, o jogador que está com o mico na mão pode usar uns truques para induzir o vizinho a pegar a carta, por exemplo, segurando-a mais alto que as outras, ou mais baixo, ou se esforçando para que ela não fique em destaque. Já o computador simplesmente pega a carta do vizinho aleatoriamente...

18.6 16.6 Classe `MaoDeMico`

Uma mão para jogar Mico requer algumas habilidades para alem das habilidades gerais de uma `Mao`. Vamos definir uma nova classe, `MaoDeMico`, que herda de `Mao` e provê um método adicional chamado `descartarCasais`:

```
class MaoDeMico(Mao):
  def descartarCasais(self):
    conta = 0
    cartasIniciais = self.cartas[:]
    for carta in cartasIniciais:
      casal = Carta(3 - carta.naipe, carta.valor)
      if casal in self.cartas:
        self.cartas.remove(carta)
        self.cartas.remove(casal)
        print "Mao %s: %s casais %s" % (self.nome,carta,casal)
        conta = conta + 1
    return conta
```

Começamos fazendo uma cópia da lista de cartas, para poder percorrer a cópia enquanto removemos cartas do original. Uma vez que `self.cartas` é modificada no laço, não queremos usá-la para controlar o percurso. Python pode ficar bem confuso se estiver percorrendo uma lista que está mudando!

Para cada carta na mão, verificamos qual é a carta que faz par com ela e vamos procurá-la. O par da carta tem o mesmo `valor` (número ou figura) e `naipe` da mesma cor. A expressão `3 - carta.naipe` transforma um paus (naipe 0) numa espadas (naipe 3) e um ouros (naipe 1) numa copas (naipe 2). Você deve analisar a fórmula até se convencer de que as operações opostas também funcionam. Se o par da carta tambem estiver na mão, ambas as cartas são removidas.

O exemplo a seguir demonstra como usar `descartarCasais`:

```
>>> jogo = JogoDeCartas()
>>> mao = MaoDeMico("fabio")
>>> jogo.baralho.distribuir([mao], 13)
>>> print mao
mão fabio contém
Ás de espadas
 2 de ouros
  7 de espadas
   8 de paus
    6 de copas
     8 de espadas
      7 de paus
       Rainha de paus
        7 de ouros
         5 de paus
          Valete de ouros
           10 de ouros
            10 de copas

>>> mao.descartarCasais()
Mão fabio: 7 de espadas faz par com 7 de paus
Mão fabio: 8 de espadas faz par com 8 de paus
Mão fabio: 10 de ouros faz par com 10 de copas
>>> print mao
Mão fabio contém
Ás de espadas
 2 de ouros
  6 de copas
   Rainha de paus
    7 de ouros
     5 de paus
      Valete de ouros
```

Observe que não existe um método `__init__` para a classe `MaoDeMico`. Ele é herdado de `Mao`.

18.7 16.7 Classe `Mico`

Agora podemos focar nossa atenção no jogo em si. `Mico` é uma subclasse de `JogoDeCartas` com um novo método chamado `jogar` que recebe uma lista de jogadores como argumento.

Já que `__init__` é herdado de `JogoDeCartas`, um novo objeto `Mico` contém um novo baralho embaralhado:

```
class Mico(JogoDeCartas):
  def jogar(self, nomes):
    # remover a Dama de paus
    self.baralho.removerCarta(Carta(0,12))

    # fazer uma mão para cada jogador
    self.maos = []
```

```
    for nome in nomes :
      self.maos.append(MaoDeMico(nome))

    # distribuir as cartas
    self.baralho.distribuir(self.maos)
    print "---------- As cartas foram dadas"
    self.exibirMaos()

    # remover casais iniciais
    casais = self.removerTodosOsCasais()
    print "---------- Os pares foram descartados, o jogo começa"
    self.exibirMaos()

    # jogar até que 25 casais se formem
    vez = 0
    numMaos = len(self.maos)
    while casais < 25:
      casais = casais + self.jogarVez(vez)
      vez = (vez + 1) % numMaos

    print "---------- Fim do jogo"
    self.exibirMaos()
```

Algumas etapas do jogo foram separadas em métodos. `removerTodosOsCasais` percorre a lista de mãos e invoca `descartarCasais` em cada uma:

```
class Mico(JogoDeCartas):
  #...
  def removerTodosOsCasais(self):
    conta = 0
    for mao in self.maos:
      conta = conta + mao.descartarCasais()
    return conta

   Como exercício, escreva ``exibirMaos`` que percorre ``self.maos`` e exibe cada mão.
```

`conta` é uma acumulador que soma o número de pares em cada mão e retorna o total.

Quando o número total de pares alcança 25, 50 cartas foram removidas das mãos, o que significa que sobrou só uma carta e o jogo chegou ao fim.

A variável `vez` mantém controle sobre de quem é a vez de jogar. Começa em 0 e incrementa de um em um; quando atinge `numMaos`, o operador módulo faz ela retornar para 0.

O método `jogarVez` recebe um argumento que indica de quem é a vez de jogar. O valor de retorno é o número de pares feitos durante essa rodada:

```
class Mico(JogoDeCartas):
  #...
  def jogarVez(self, i):
    if self.maos[i].estahVazia():
      return 0
    vizinho = self.buscarVizinho(i)
    novaCarta = self.maos[vizinho].pegarCarta()
    self.maos[i].adicionarCarta(novaCarta)
    print "Mao", self.maos[i].nome, "pegou", novaCarta
    conta = self.maos[i].descartarCasais()
    self.maos[i].embaralhar()
    return conta
```

Se a mão de um jogador estiver vazia, ele está fora do jogo, então, ele não faz nada e retorna 0.

Do contrário, uma jogada consiste em achar o primeiro jogador à esquerda que tenha cartas, pegar uma carta dele, e tentar fazer pares. Antes de retornar, as cartas na mão são embaralhadas, para que a escolha do próximo jogador seja aleatória.

O método `buscarVizinho` começa com o jogador imediatamente à esquerda e continua ao redor da mesa até encontrar um jogador que ainda tenha cartas:

```
class Mico(JogoDeCartas):
  #...
  def buscarVizinho(self, i):
    numMaos = len(self.maos)
    for next in range(1,numMaos):
      vizinho = (i + next) % numMaos
      if not self.maos[vizinho].estahVazia():
        return vizinho
```

Se `buscarVizinho` alguma vez circulasse pela mesa sem encontrar cartas, retornaria `None` e causaria um erro em outra parte do programa. Felizmente, podemos provar que isso nunca vai acontecer (desde que o fim do jogo seja detectado corretamente).

Não mencionamos o método `exibirBaralhos`. Esse você mesmo pode escrever.

A saída a seguir é produto de uma forma reduzida do jogo, onde apenas as 15 cartas mais altas do baralho (do 10 para cima) foram dadas, para três jogadores. Com esse baralho reduzido, a jogada pára depois que 7 combinações foram feitas, ao invés de 25:

```
>>> import cartas
>>> jogo = cartas.Mico()
>>> jogo.jogar(["Alice","Jair","Clara"])
---------- As cartas foram dadas
Mão Alice contém
Rei de copas
 Valete de paus
  Rainha de espadas
   Rei de espadas
    10 de ouros

Mão Jair contém
Rainha de copas
 Valete de espadas
  Valete de copas
   Rei de ouros
    Rainha de ouros

Mão Clara contém
Valete of ouros
 Rei de paus
  10 de espadas
   10 de copas
    10 de paus

Mão Jair: Dama de copas faz par com Dama de ouros
Mão Clara: 10 de espadas faz par com 10 de paus
---------- Os pares foram descartados, o jogo começa
Mão Alice contém
Rei de copas
 Valete de paus
  Rainha de espadas
```

```
   Rei de espadas
    10 de ouros

Mão Jair contém
Valete de espadas
 Valete de copas
  Rei de ouros

Mão Clara contém
Valete de ouros
 Rei de paus
  10 de copas

Mão Alice pegou o Rei de ouros
Mão Alice: Rei de copas faz par com Rei de ouros
Mão Jair pegou 10 de copas
Mão Clara pegou Valete de paus
Mão Alice pegou Valete de copas
Mão Jair pegou Valete de ouros
Mão Clara pegou Dama de espadas
Mão Alice pegou Valete de ouros
Mão Alice: Valete de copas faz par com Valete de ouros
Mão Jair pegou Rei de paus
Mão Clara pegou Rei de espadas
Mão Alice pegou 10 de copas
Mão Alice: 10 de ouros faz par com 10 de copas
Mão Jair pegou Dama de espadas
Mão Clara pegou Valete de espadas
Mão Clara: Valete de paus faz par com Valete de espadas
Mão Jair pegou Rei de espadas
Mão Jeff: Rei de paus faz par com Rei de espadas
---------- Fim do jogo
Mão Alice está vazia

Mão Jair contém
Rainha de espadas

Mão Clara está vazia
```

Então, o Jair perdeu.

18.8 16.8 Glossário

herança (*inheritance*) Habilidade de definir uma nova classe que é a versão modificada de uma classe definida anteriormente.

classe mãe (*parent class*) A classe de quem a classe filha herda.

classe filho (*child class*) Um nova classe criada herdando de uma classe existente; também chamada de "subclasse".

CAPÍTULO 19

Capítulo 17: Listas encadeadas

Tópicos

19.1 17.1 Referências Embutidas

Nós temos visto exemplos de atributos que referenciam outros objetos, que são chamados **referências embutidas** (veja a Seção 12.8). Uma estrutura de dados comum, a **lista ligada**, tira vantagem desta característica.

Listas ligadas são constituídas de **nós** (nodos), onde cada nó contém uma referência para o próximo nó na lista. Além disto, cada nó contém uma unidade de dados chamada a **carga**.

Uma lista ligada é considerada uma **estrutura de dados recorrente** porque ela tem uma definição recorrente.

Uma lista ligada é:

- Uma lista vazia, representada por *None*, ou
- Um nó que contém um objeto carga e uma referência para uma lista ligada.

Estruturas de dados recorrentes são adequadas para métodos recorrentes.

19.2 17.2 A classe `No` (Node)

Como é usual quando se escreve uma nova classe, nós começaremos com os métodos de inicialização e `__str__` de modo que podemos testar o mecanismo básico de se criar e mostrar o novo tipo:

```
class No:
  def __init__(self, carga=None, proximo=None):
    self.carga = carga
    self.proximo = proximo

  def __str__(self):
    return str(self.carga)
```

Como de costume, os parâmetros para o método de inicialização são opcionais. Por omissão (*default*), ambos, a carga e a ligação, `proximo`, são definidas como `None`.

A representação string de um nó é simplesmente a representação string da carga. Como qualquer valor pode ser passado para a função `str`, nós podemos armazenar qualquer valor em uma lista.

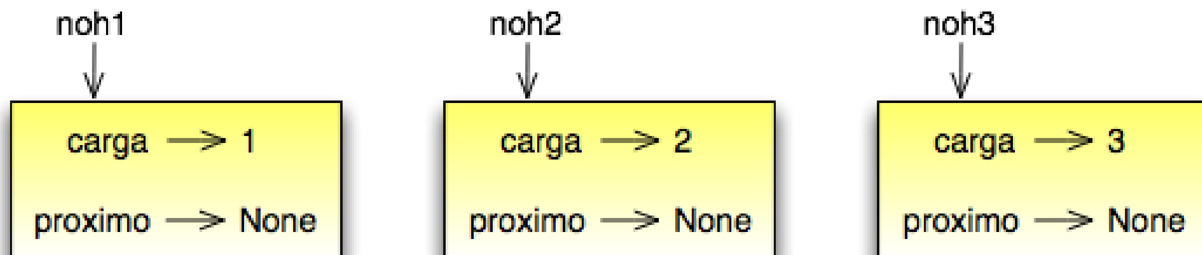
Para testar a implementação até agora, nós criamos um `No` e o imprimimos:

```
>>> no = No("teste")
>>> print no
teste
```

Para ficar interessante, nós precisamos uma lista com mais do que um nó:

```
>>> no1 = No(1)
>>> no2 = No(2)
>>> no3 = No(3)
```

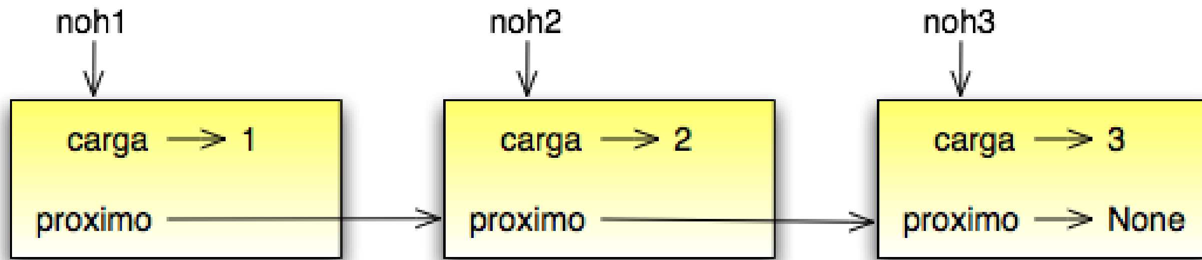
Este código cria três nós, mas nós ainda não temos uma lista ainda porque os nós não estão **ligados**. O diagrama de estado é parecido com este:



Para ligar os nós, temos que fazer o primeiro nó da lista referir ao segundo e o segundo nó referir ao terceiro:

```
>>> no1.proximo = no2
>>> no2.proximo = no3
```

A referência do terceiro nó é `None`, que indica que ele é o final da lista. Agora o diagrama de estado se parece com:



Agora você sabe como criar nós e ligá-los em uma lista. O que pode estar menos claro neste ponto é por quê.

19.3 17.3 Listas como Coleções

Listas são úteis porque elas provêm um modo de montar múltiplos objetos em uma única entidade, algumas vezes chamada uma **coleção**. No exemplo, o primeiro nó da lista serve como uma referência para toda a lista.

Para passar uma lista como um parâmetro, você apenas tem que passar uma referência ao primeiro nó. Por exemplo, a função `imprimeLista` toma um único nó como um argumento. Iniciando com o cabeça da lista, ela imprime cada nó até que chegue ao fim:

```
def imprimeLista(no):
  while no:
    print no,
    no = no.proximo
  print
```

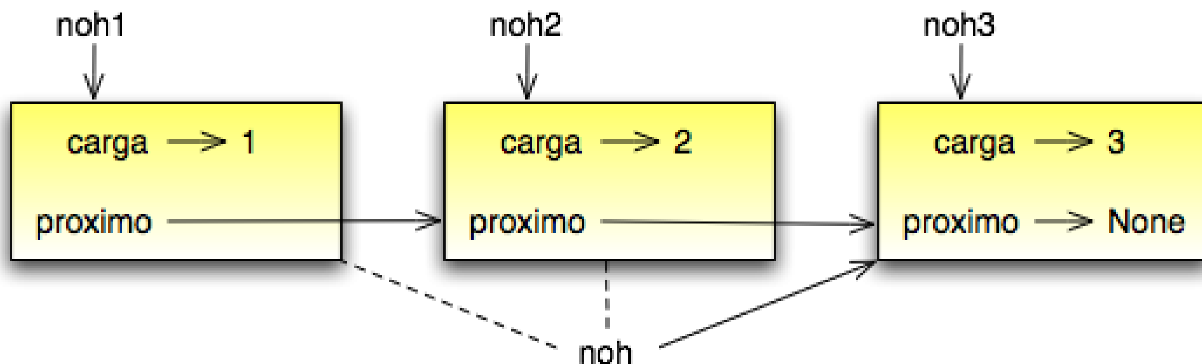
Para chamar este método, nós passamos uma referência ao primeiro no:

```
>>> imprimeLista(no1)
1 2 3
```

Dentro de `imprimeLista` nós temos uma referência para o primeiro nó da lista, mas não há variáveis que refiram aos outros nós. Nós temos que usar o valor `proximo` de cada nó para alcançar o próximo nó.

Para percorrer uma lista ligada, é comum usar uma variável laço como `no` para referir a cada um dos nós sucessivamente.

Este diagrama mostra o valor de `lista` e os valores que `no` assume:



Por convenção, listas são freqüentemente impressas em braquetes com vírgulas entre os elementos, como em [1, 2, 3]. Como um exercício, modifique `imprimeLista` *para que ela gere uma saída neste formato.*

19.4 17.4 Listas e Recorrência

É natural expressar muitas operações de listas utilizando métodos recorrentes. Por exemplo, o seguinte é um algoritmo recorrente para imprimir uma lista de trás para frente.

1. Separe a lista em dois pedaços: o primeiro nó (chamado a cabeça); e o resto (chamado o rabo).
2. Imprima o rabo de trás para frente.
3. Imprima a cabeça.

Logicamente, o Passo 2, a chamada recorrente, assume que nós temos um modo de imprimir a lista de trás para frente. Mas se nós assumimos que a chamada recorrente funciona – o passo de fé – então podemos nos convencer de que o algoritmo funciona.

Tudo o que precisamos são um caso base e um modo de provar que para qualquer lista, nós iremos, ao final, chegar no caso base. Dada a definição recorrente de uma lista, um caso base natural é a lista vazia, representada por `None`:

```
def imprimeDeTrasParaFrente(lista):
  if lista == None : return
  cabeca = lista
  rabo = lista.proximo
  imprimeDeTrasParaFrente(rabo)
  print cabeca,
```

A primeira linha trata o caso base fazendo nada. As próximas duas linhas dividem a lista em `cabeca` e `rabo`. As duas últimas linhas imprimem a lista. A vírgula no final da última linha impede o Python de imprimir uma nova linha após cada nó.

Nós invocamos este método como invocamos o `imprimeLista`:

```
>>> imprimeDeTrasParaFrente(no1)
3 2 1
```

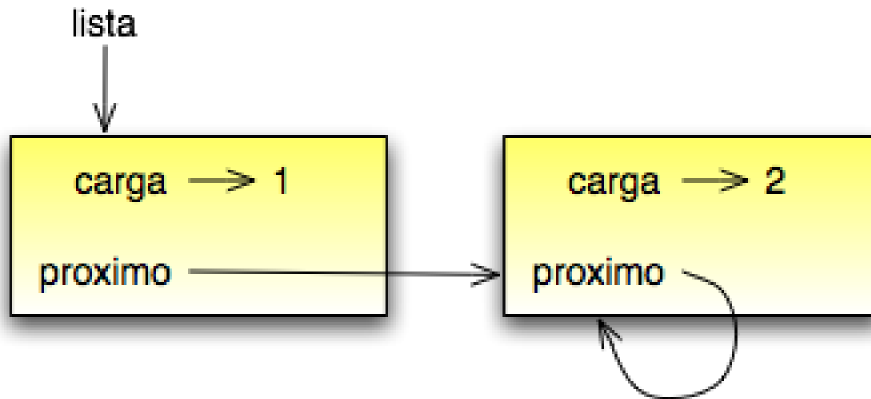
O resultado é a lista de trás para frente.

Você pode se perguntar por quê `imprimeLista` e `imprimeDeTrasParaFrente` são funções e não métodos da classe `No`. A razão é que nós queremos usar `None` para representa a lista vazia e não é legal invocar um método sobre `None`. Esta limitação torna complicado escrever código de manipulação de lista em estilo orientado a objeto limpo.

Podemos provar que `imprimeDeTrasParaFrente` sempre termina? Em outras palavras, irá ela sempre atingir o caso base? De fato, a resposta é não. Algumas listas farão este método falhar.

19.5 17.5 Listas Infinitas

Não há nada que impeça um nó de referenciar de volta um nó anterior na lista, incluindo ele mesmo. Por exemplo, esta figura mostra uma lista com dois nós, um dos quais refere-se a si mesmo:



Se nós invocarmos `imprimeLista` nesta lista, ele ficará em laço para sempre. Se nós invocarmos `imprimeDeTrasParaFrente`, ele recorrerá infinitamente. Este tipo de comportamento torna as listas infinitas difíceis de se lidar.

A despeito disto, elas ocasionalmente são úteis. Por exemplo, podemos representar um número como uma lista de dígitos e usar uma lista infinita para representar uma fração repetente.

Mesmo assim, é problemático que não possamos provar que `imprimeLista` e `imprimeDeTrasParaFrente` terminem. O melhor que podemos fazer é a afirmação hipotética, "Se a lista não contém laços, então este método terminará." Este tipo de hipótese é chamado uma **pré-condição**. Ele impõe uma limitação sobre um dos parâmetros e descreve o comportamento do método se a limitação é satisfeita. Você verá mais exemplos em breve.

19.6 17.6 O Teorema da Ambigüidade Fundamental

Uma parte de `imprimeDeTrasParaFrente` pode ter gerado surpresa:

```
cabeca = lista
rabo = lista.proximo
```

Após a primeira atribuição, `cabeca` e `lista` têm o mesmo tipo e o mesmo valor. Então por que nós criamos uma nova variável?

A razão é que as duas variáveis têm diferentes papéis. Quando pensamos em `cabeca`, pensamos como uma referência a um único nó, e quando pensamos em `lista` o fazemos como uma referência ao primeiro nó da lista. Estes "papéis" não são parte do programa; eles estão na mente do programador.

Em geral não podemos dizer olhando para o programa qual o papel que uma variável tem. Esta ambigüidade pode ser útil, mas também pode tornar os programas difíceis de serem lidos. Usamos freqüentemente nomes de variáveis como `no` e `lista` para documentar como pretendemos usar uma variável e algumas vezes criamos variáveis adicionais para remover a ambigüidade.

Poderíamos ter escrito `imprimeDeTrasParaFrente` sem `cabeca` e `rabo`, que a tornaria mais concisa mas possivelmente menos clara:

```
def imprimeDeTrasParaFrente(lista):
  if lista == None : return
  imprimeDeTrasParaFrente(lista.proximo)
  print lista,
```

Olhando para as duas chamadas de função, temos que lembrar que `imprimeDeTrasParaFrente` trata seu argumento como uma coleção e `print` trata seu argumento como um objeto único.

O **teorema da ambigüidade fundamental** descreve a ambigüidade que é inerente à referência a um nó:

Uma variável que refere a um nó pode tratar o nó como um objeto único ou como o primeiro em uma lista de nós.

19.7 17.7 Modificando Listas

Existem duas maneiras de se modificar uma lista ligada. Obviamente, podemos modificar a carga dos nós, mas as operações mais interessantes são aquelas que adicionam, removem ou reordenam os nós.

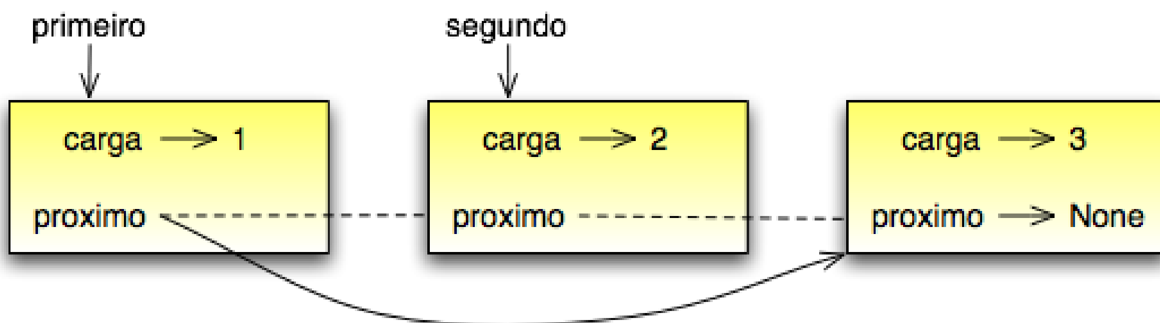
Como um exemplo, vamos escrever um método que remove o segundo nó na lista e retorna uma referência ao nó removido:

```
def removeSegundo(lista):
  if lista == None : return
  primeiro = lista
  segundo = lista.proximo
  # faz o primeiro no referir ao terceiro
  primeiro.proximo = segundo.proximo
  # separa o segundo no do resto da lista
  segundo.proximo = None
  return segundo
```

Novamente, estamos usando variáveis temporárias para tornar o código mais fácil de ser lido. Aqui está como usar este método:

```
>>> imprimeLista(no1)
1 2 3
>>> removido = removeSegundo(no1)
>>> imprimeLista(removido)
2
>>> imprimeLista(no1)
1 3
```

Este diagrama de estado mostra o efeito da operação:



O que acontece se você invocar este método e passar uma lista com somente um elemento (um **singleton**)? O que acontece se você passar a lista vazia como um argumento? Existe uma pré-condição para este método? Se houver, corrija o método para tratar uma violação da pré-condição de modo razoável.

19.8 17.8 Envoltórios e Ajudadores

Freqüentemente é útil dividir uma operação de lista em dois métodos. Por exemplo, para imprimir uma lista de trás para frente no formato convencional de lista [3, 2, 1], podemos usar o método `imprimeDeTrasParaFrente` para

imprimir 3, 2, mas queremos um metodo separado para imprimir os braquetes e o primeiro nó. Vamos chamá-lo de `imprimeDeTrasParaFrenteLegal`:

```
def imprimeDeTrasParaFrenteLegal(lista):
  print "[",
  if lista != None :
    cabeca = lista
    rabo = lista.proximo
    imprimeDeTrasParaFrente(rabo)
    print cabeca,
  print "]",
```

Novamente, é uma boa idéia verificar métodos como este para ver se eles funcionam com casos especiais como uma lista vazia ou um singleton.

Quando usamos este método em algum lugar no programa, invocamos `imprimeDeTrasParaFrenteLegal` diretamente, e ele invoca `imprimeDeTrasParaFrente` por nós. Neste sentido, `imprimeDeTrasParaFrenteLegal` atua como um **envoltório**, e usa `imprimeDeTrasParaFrente` como um **ajudador**.

19.9 17.9 A Classe `ListaLigada`

Existem alguns problemas sutis com o modo que implementamos listas. Em um inverso de causa e efeito, proporemos uma implementação alternativa primeiro e então explicaremos qual problema ela resolve.

Primeiro, criaremos uma nova classe chamada `ListaLigada`. Seus atributos são um inteiro que contém o comprimento da lista e uma referência para o primeiro nó. Objetos do tipo `ListaLigada` servem como cabos (*handles*) para se manipular listas de objetos `No`:

```
class ListaLigada:
  def __init__(self):
    self.comprimento = 0
    self.cabeca = None
```

Uma coisa legal acerca da classe `ListaLigada` é que ela provê um lugar natural para se colocar funções envoltórias como `imprimeDeTrasParaFrenteLegal`, que podemos transformar em um método da classe `ListaLigada`:

```
class ListaLigada:
  ...
  def imprimeDeTrasParaFrente(self):
    print "[",
    if self.cabeca != None :
      self.cabeca.imprimeDeTrasParaFrente()
    print "]",


class No:
  ...
  def imprimeDeTrasParaFrente(self):
    if self.proximo != None:
      rabo = self.proximo
      rabo.imprimeDeTrasParaFrente()
    print self.carga,
```

Apenas para tornar as coisas confusas, mudamos o nome de `imprimeDeTrasParaFrenteLegal`. Agora existem dois métodos chamados `imprimeDeTrasParaFrente`: um na classe `No` (o ajudador); e um na classe `ListaLigada``(o envoltório). Quano o envoltório invoca`

``self.cabeca.imprimeDeTrasParaFrente, ele está invocando o ajudador, porque self.cabeca é um objeto No.

Outro benefício da classe ListaLigada é que ela torna mais fácil adicionar e remover o primeiro elemento de uma lista. Por exemplo, adicionaPrimeiro é um método para ListaLigada; ele toma um item de carga como argumento e o coloca no início da lista:

```
class ListaLigada:
  ...
  def adicionaPrimeiro(self, carga):
    no = No(carga)
    no.proximo = self.cabeca
    self.cabeca = no
    self.comprimento = self.comprimento + 1
```

Como de costume, você deve conferir códigos como este para ver se eles tratam os casos especiais. Por exemplo, o que acontece se a lista está inicialmente vazia?

19.10 17.10 Invariantes

Algumas listas são "bem formadas"; outras não o são. Por exemplo, se uma lista contém um laço, ela fará muitos de nossos métodos falharem, de modo que podemos querer requerer que listas não contenham laços. Outro requerimento é que o valor de comprimento no objeto ListaLigada seja igual ao número real de nós da lista.

Requerimentos como estes são chamados de **invariantes** porque, idealmente, eles deveriam ser verdade para cada objeto o tempo todo. Especificar invariantes para objetos é um prática de programação útil porque torna mais fácil provar a correção do código, verificar a integridade das estruturas de dados e detectar erros.

Uma coisa que algumas vezes é confusa acerca de invariantes é que existem momentos em que eles são violados. Por exemplo, no meio de adicionaPrimeiro, após termos adicionado o nó mas antes de termos incrementado comprimento, o invariante é violado. Este tipo de violação é aceitável; de fato, é freqüentemente impossível modificar um objeto sem violar um invariante por, no mínimo, um pequeno instante. Normalmente, requeremos que cada método que viola um invariante deve restaurar este invariante.

Se há qualquer aumento significativo de código no qual o invariante é violado, é importante tornar isto claro nos comentários, de modo que nenhuma operação seja feita que dependa daquele invariante.

19.11 17.11 Glossário

referência embutida (*embedded reference*) Uma referência armazenada/associada em/a um atributo de um objeto.

lista ligada (*linked list*) Uma estrutura de dados que implementa uma coleção usando uma sequência de nós ligados.

nó ou nodo (*node*) Um elemento de uma lista, usualmente implementado como um objeto que contém uma referência para outro objeto do mesmo tipo.

carga (*cargo*) Um item de dado contido em um nó.

ligação (*link*) Uma referência embutida usada para ligar/conectar um objeto a outro.

pré-condição (*precondition*) Uma asserção que precisa/deve ser verdadeira para que um método trabalhe corretamente.

teorema da ambigüidade fundamental (*fundamental ambiguity theorem*) Uma referência para um nó de uma lista pode ser tratada como um objeto único ou como o primeiro em uma lista de nós.

singleton (*singleton*) Uma lista ligada com somente um nó.

envoltório (*wrapper*) Um método que atua como um intermediário (*middleman*) entre um chamador e um método ajudador (*helper*), freqüentemente tornando a invocação do método mais fácil ou menos propensa a erros.

ajudador (*helper*) Um método que não é invocado diretamente pelo chamador (*caller*) mas é usado por outro método para realizar parte de uma operação.

invariante (*invariant*) Uma asserção que deveria ser verdadeira sempre para um objeto (exceto talvez enquanto o objeto estiver sendo modificado).

Capítulo 18: Pilhas

Tópicos

20.1 18.1 Tipos abstratos de dados

Os tipos de dados que você viu até agora são todos concretos, no sentido que nós especificamos completamente como eles são implementados. Por exemplo, a classe Card(*XXX ver como foi traduzido*) representa uma carta utilizando dois inteiros. Como discutimos no momento, esta não é a única maneira de representar uma carta; existem muitas implementações alternativas.

Um **tipo abstrato de dado**, ou TAD, especifica um conjunto de operações (ou métodos) e a semântica das operações (o que elas fazem), mas não especifica a implementação das operações. Isto é o que o faz abstrato.

Por que isto é útil?

- Simplifica a tarefa dde especificar um algoritmo se você pode XXXdenotar(*denote*) as operações que você precisa sem ter que pensar, ao mesmo tempo, como as operações são executadas.
- Uma vez que existem geralmente muitas maneiras de implementar um TAD, pode ser útil escrever um algritmo que pode ser usado com qualquer das possíveis implementações.
- TADs bastante conhecidos, como o TAD Pilha deste capítulo, já estão implementados em bibliotecas padrão, então eles podem ser escritos uma vez e usado por muitos programadores.
- As operações em TADs provêm uma linguagem de alto nível comum para especificar e falar sobre algoritmos.

Quando falamos sobre TADs, geralmente distinguimos o código que usa o TAD, chamado **cliente**, do código que implementa o TAD, chamado código **fornecedor**.

20.2 18.2 O TAD Pilha

Neste capítulo, iremos olhar um TAD comum, a **pilha**. Uma pilha é uma coleção, ou seja, é uma estrutura de dados que contei múltiplos elementos. Outras coleções que vimos incluem dicionários e listas.

Um TAd é definido pelo conjunto de operações que podem ser executadas nele, que é chamado **interface**. A interface para uma pilha consiste nestas operações:

`__init__` : Inicializa uma nova pilha vazia.

`push` : Adiciona um novo item na pilha

`pop` : Remove um ítem da pilha e o retorna, O ítem que é retornadao é sempre o último adicionado.

`isEmpty` : Verifica se a pilha está vazia.

Uma às vezes é chamada uma estrutura de dados "last in, first out" ou LIFO, porque o último ítem adicionad é o primeiro a ser removido.

20.3 18.3 Implementando pilhas com listas de Python

As operações de lista que Python oferecem são similares às operações que definem uma pilha. A interface não é exatamente o que se supõe ser, mas podemos escrever um código para traduzir do TAD Pilha para as operações nativas.

Este código é chamado uma **implementação** do TAD Pilha. Geralmente, uma implementaçõa é um conjunto de métodos que satisfazem os requisitos sintáticos e semânticos de uma interface.

Aqui está uma implementação do TAD Pilha que usa uma lista do Python:

```
class Stack :
  def __init__(self) :
    self.items = []

  def push(self, item) :
    self.items.apend(item)

  def pop(self) :
    return self.items.pop()

  def isEmpty(self) :
    return (self.items == [])
```

Um objeto `Stack` contém um atributo chamado `items` que é uma lista de ítens na pilha. O método de inicialização define items como uma lista vazia.

Para adicionar um novo ítem na pilha, `push` o coloca em `items`. Para remover um ítem da pilha, pop usa o método de lista homônimo[1] para remover e retornar um último ítem da lista.

Finalmente, para verificar se a pilha está vazia, `isEmpty` comprara items a uma lista vazia.

Uma implementação como esta, na qual os métodos consistem de simples invocações de métodos existentes, é chamado **revestimento**. Na vida real, revestimento é uma fina camada de madeira de boa qualidade usado em XXX*furniture-making* para esconder madeira de menor qualidade embaixo. Cientistas da Computação usam esta metáfora para descrever um pequeno trecho de código que esconde os detalhes de uma implementação e fornece uma interface mais simples, ou mais padronizada.

20.4 18.4 Empilhando e desempilhando

Uma pilha é uma **estrutura de dados genérica**, o que significa que podemos adicionar qualquer tipo de ítem a ela. O exemplo a seguir empilha dois inteiros e uma XXXstring na pilha:

```
>>> s = Stack()
>>> s.push(54)
>>> s.push(45)
>>> s.push("+")
```

Podemos usar `isEmpty` e pop para remover e imprimir todos os ítens da pilha:

while not s.isEmpty() : priint s.pop()

A saída é + 45 54. Em outras palavras, usamos a pilha para imprimir os ítens ao contrário! Sabidamente, este não é o formato padrão de imprimir uma lista, mas, usando uma pilha, foi notavelmente fácil de fazer.

Você deve comparar este trecho de código com a implementação de *printBackward* na seção 17.4. Existe um paralelo natura entre a versão recursiva de *printBackward* e o algoritmo de pilha aqui. A diferenã é que printBackward usa a pilha de execução para XXXmanter a trilha(keep track) dos nós enquanto percorre a lista, e então imprime-a no retorno da recursão. o algoritmo de pilha faz a mesma coisa, exceto que usa o objeto Stack ao invés da pilha de execução.

20.5 18.5 Usando uma pilha para avaliar expressões pós-fixas

Em muitas linguagens de programação, expressões matemáticas são executadas com o poerador entre os roid operandos, como em 1 + 2. Este formato é chamado notação **infixa**. Uma alternativa usada por algumas calculadoras é chamada notação **pós-fixa**. Em notação pós-fixa, o operador segue os operandos, como em 1 2 +.

A razão pela qual a notação pós-fixa é útil algumas vezes é que existe uma maneira natural de avaliar uma expressão pós-fixa usando uma pilha:

- começando no início da expressão, peque um termo (operador ou operando) de cada vez.
- Se o termo é um operando, empilhe-o
- Se o termo é um operador, desempilhe dois operandos, execute a operação neles, e empilhe o resultado.
- Quando chegar ao fim da expressão, deve existir exatamente um operando sobrando na pilha. Este operando é o resultado.
- Como um exercício, aplique este algoritmo à expressão 1 2 + 3 * .

Este exemplo demonstra uma das vantagens da notação pós-fixa - não é necessário usar parênteses para controlar a ordem das operações. Para ter o mesmo resultado em notação infixa, deveríamos escrever (1 + 2) * 3.

- Como um exercício, escreva a expressão pós-fixa que é equivalente a 1 + 2 * 3.

20.6 18.6 Análise sintática

Para implementar o algoritmo anterior, necessitamos estar prontos para percorrer uma string e quebrá-la em operandos e operadores. Este processó é um exemplo de **XXXparsing**, e o resultado - os pedaços da string - são chamados **XXXtokens**. Você deve lembrar estas palavras do capítulo 1.

Python fornece um método split nos módulos string e re (expressões regulares). A função `string.split` separa uma string numa lista usando um único caracter como **delimitador**. Por exemplo:

```
>>> import string
>>> string.split ("Now is the time", " ")
['Now', 'is', 'the', 'time']
```

Neste caso, o delimitador é o caracter de espaço, então a string é dividida a cada espaço.

A função `re.split` é mais poderosa, permitindo-nos fornecer uma expresão regular ao invés de um delimitador. Uma expressão regular é uma maneira de especificar um conjunto de strings. Por exemplo, [A-z] é o conjunto de todas as letras e [0-9] é o conjunto de todos os dígitos. O operador ^nega um conunto, então [^0-9] é o conjunto de tudo o que não é número, que é exatamente o que queremos para dividir expressões pós-fixas.

```
>>> import re
>>> re.split ("[^0-9]", "123+456*/")
['123', '+', '456', '*', '', '/', ' ']
```

Note que a ordem dos argumentos é diferente de string.split, o delimitador vem antes da string.

A lista resultante inclui os operandos `123` e `456`, e os operadores `*` e `/`. Também inclui duas strings vazias que são inseridas depois dos operadores.

20.7 18.7 Avaliando em pós-fixo.

Para avaliar uma expressão pós-fixa, usaremos o parser da seção anterior e o algoritmo da seção anterior a ela. Para manter as coisas simples, começaremos com um avaliador que implementa somente os operadores + e .

```
def evalPostfix (expr) :
  import re
  tokenList = re.split ("([^0-9])", expr)
  stack = Stack()
  for token in tokenList
    if token == '' or token = ' ' :
      continue
    if token == '+' :
      sum = stack.pop() + stack.pop()
      stack.push(sum)
    if token == '*' :
      product = stack.pop() * stack.pop()
      stack.push(product)
    else:
      stack.push(int(token))
  return stack.pop()
```

A primeira condição cuida de espaços e strings vazias. As duas próximas condições manipulam os operadores. Nós assumimos, agora que qualquer coisa é um operador. É claro, seria melhor chegar por entrada errônea e enviar uma mensagem de erro, mas faremos isto depois.

Vamos testá-lo avaliando a forma pós-fixa de `(56 + 47) * 2`

```
>>> print evalPostfix("56 47 + 2 *")
206
```

XXXthat's close enough

18.8 Clientes de fornecedores.

Um dos objetivos de um TAD é separar os interesses do fornecedor, quem escreve o código que implementa o TAD, e o cliente, que usa o TAD. O fornecedor tem que se preocupar apenas se a implementação está correta - de acordo com a especificação do TAD - e não como ele será usado.

Inversamente, o cliente assume que a implementação do TAD está correta e não se preocupa com os detalhes. Quando você está usando um tipo nativo do Python, tem o luxo de pensar exclusivamente como um cliente.

É claro, quanto você implementa um TAD, você também tem que escrever código cliente para testá-lo. Neste caso, você faz os dois papéis, o que pode ser confuso. Você deve fazer algum esforçõ para manter a trilha do papel que está fazendo a cada momento.

20.7.1 18.9 Glossário

tipo abstrato de dados (TAD) (*abstract data type(ADT)*): Um tipo de dado(geralmente uma coleção de objetos) que é definidopor um conjunto de operações, que podem ser implementadas de várias maneiras.

interface (*interface*): É o conjunto de operações que definem um TDA.

implementação (*implementation*): Código que satisfaz(preenche?) os requisitos sintáticos e semânticos de uma interface.

cliente (*client*): Um programa (ou o programador que o escreveu) que faz utilização de um TDA.

fornecedor (*provider*): Código (ou o programador que o escreveu) que implementa um TDA.

revestimento (*veneer*): Definição de classe que implementa um TDA com definições de métodos que são chamadas a outros métodos, às vezes com pequenas modificações. A lâmina faz um trabalho insignificante, mas melhora ou padroniza a interface dada ao cliente.

estrutura de dados genérica (*generic data structure*): Tipo de estrutura de dados que contem data de um tipo qualquer(tipo genérico).

infixa (*infix*): Notação matemática em que os operadores se situam entre os operandos.

pós-fixa (*postfix*): Notação matemática em que os operadores se situam após os operandos.

XXX parse (*parse*): Ler um conjunto de caracteres(string de caracteres) ou tokens(símbolos atômicos) e analisar sua estrutura gramatical.

XXX token (*token*): Conjunto de caracteres que são tratados como uma unidade atômica para fins de análise gramatical, como as palavras na linguagem natural.

delimitador (*delimiter*): Um caracter que é utilizado para separar os símbolos atômicos(tokens), como a pontuação na linguagem natural.

Capítulo 19: Filas

Este capítulo apresenta dois TDAs: Fila e Fila por Prioridade. Na nossa vida diária, fila é um alinhamento de consumidores aguardando algum tipo de serviço. Na maioria dos casos, o primeiro da fila é o primeiro a ser atendido. Mas há exceções. No aeroporto, passageiros cujo vôo vai decolar logo, às vezes são chamados primeiro ao balcão do check-in, mesmo que estejam no meio da fila. No supermercado, é comum na fila do caixa alguém deixar passar na frente uma pessoa que chega à fila só com um ou dois produtos na mão.

A regra que determina quem é o próximo da fila chama-se política de enfileiramento. A política de enfileiramento mais simples chama-se FIFO, sigla de *first-in-first-out*: primeiro a entrar, primeiro a sair. A política de enfileiramento mais geral é o enfileiramento por prioridade, em que se atribui uma prioridade a cada pessoa da fila e a que tiver maior prioridade vai primeiro, independente da sua ordem de chegada. Dizemos que essa é a política mais geral de todas, porque a prioridade pode ser baseada em qualquer coisa: hora de partida do vôo; quantos produtos a pessoa vai passar pelo caixa; o grau de prestígio da pessoa. É claro que nem todas as políticas de enfileiramento são "justas", mas o que é justo depende do ponto de vista.

O TDA Fila e o TDA Fila por Prioridade têm o mesmo conjunto de operações. A diferença está na semântica das operações: a fila usa a política FIFO; e a fila por prioridade (como o próprio nome sugere) usa a política de enfileiramento por prioridade.

21.1 19.1 Um TDA Fila

O TDA Fila é definido pelas seguintes operações:

__init__ Inicializar uma nova fila vazia.

insert Adicionar um novo item à fila.

remove Remover e retornar um item da fila. O item retornado é o que foi adicionado primeiro.

isEmpty Checar se a fila está vazia.

21.2 19.2 Fila encadeada

A primeira implementação que vamos ver de um TDA Fila chama-se **fila encadeada** porque é feita de objetos `Nós` encadeados. A definição da classe é a seguinte:

```
class Queue:
  def __init__(self):
    self.length = 0
    self.head = None

  def isEmpty(self):
    return (self.length == 0)

  def insert(self, cargo):
    node = Node(cargo)
    node.next = None
    if self.head == None:
      # if list is empty the new node goes first
      self.head = node
    else:
      # find the last node in the list
      last = self.head
      while last.next: last = last.next
      # append the new node
      last.next = node
    self.length = self.length + 1

  def remove(self):
    cargo = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    return cargo
```

Os métodos `isEmpty` e `remove` são idênticos aos métodos `isEmpty` e `removeFirst` de `LinkedList`. O método `insert` é novo e um pouco mais complicado.

Queremos inserir novos itens no fim da lista. Se a fila estiver vazia, basta fazer `head` apontar ao novo nó. Se não, percorremos a lista até o último nó e lá penduramos o novo nó. É possível identificar o último nó porque o seu atributo `next` é `None`.

Existem duas invariantes para um objeto `Fila` bem formado: o atributo length deve ser o número de nós na fila, e o último nó deve ter seu atributo `next` igual a `None`. Estude o método até ficar convencido de que ele preserva ambas invariantes.

21.3 19.3 Características de performance

Quando invocamos um método, normalmente não estamos preocupados com os detalhes da sua implementação. Porém, há um certo "detalhe" que pode ser bom conhecer: as características de performance do método. Quanto tempo leva, e como o tempo de execução muda à medida em que aumenta o número de itens da coleção?

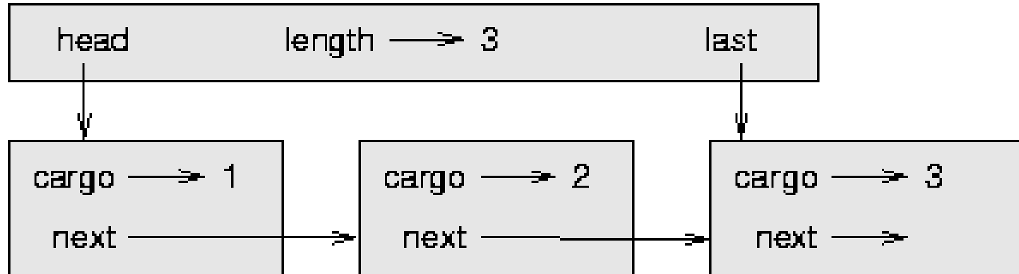
Primeiro, olhe para `remove`. Não há laços ou chamadas de função aqui, o que sugere que o tempo de execução desse método é sempre o mesmo, toda vez. Um método assim é chamado de operação de **tempo constante**. Na verdade, o método pode ser ligeiramente mais rápido quando a lista está vazia, uma vez que ele pula o corpo da condicional, mas essa diferença não é significativa. XXX: o condicional só aparece na re-implementação do método na classe ImprovedQueue, p.200; essa inconsistência pode ser conferida também nas páginas 198-199 do livro original (PDF e impresso).

A performance de `insert` é muito diferente. No caso geral, temos de percorrer a lista para achar o último elemento.

Este percurso leva um tempo proporcional à extensão da lista. Uma vez que o tempo de execução é uma função linear da extensão, dizemos que este método opera em **tempo linear**. Isso é bem ruim, se comparado com o tempo constante.

21.4 19.4 Fila encadeada aprimorada

Queremos uma implementação do TDA Fila que possa realizar todas as operações em tempo constante. Uma maneira de fazer isso é modificar a classe Fila, de modo que ela mantenha a referência tanto ao primeiro quanto ao último nó, como mostra a figura:



A implementação de `ImprovedQueue` tem essa cara:

```
class ImprovedQueue:
  def __init__(self):
    self.length = 0
    self.head   = None
    self.last   = None

  def isEmpty(self):
    return (self.length == 0)
```

Até agora, a única mudança é o atributo `last`. Ele é usado nos métodos `insert` e `remove`:

```
class ImprovedQueue:
  # ...
  def insert(self, cargo):
    node = Node(cargo)
    node.next = None
    if self.length == 0:
      # if list is empty, the new node is head and last
      self.head = self.last = node
    else:
      # find the last node
      last = self.last
      # append the new node
      last.next = node
      self.last = node
    self.length = self.length + 1
```

Uma vez que `last` não perde de vista o ultimo nó, não é necessário buscá-lo. Como resultado, esse método tem tempo constante.

Mas essa rapidez tem preço. É preciso adicionar um caso especial a `remove`, para configurar `last` para `None` quando o ultimo nó é removido:

```
class ImprovedQueue:
  #...
  def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    if self.length == 0:
```

```
      self.last = None
    return cargo
```

Essa implementação é mais complicada que a primeira, e mais difícil de se demonstrar que está correta. A vantagem é que o objetivo foi atingido – tanto `insert` quanto `remove`` são operações de tempo constante.

> Como exercício, escreva uma implementação do TDA Fila usando uma lista nativa do Python. Compare a performance dessa implementação com a de `ImprovedQueue`, para filas de diversos comprimentos.

21.5 19.5 Fila por prioridade

O TDA Fila por Prioridade tem a mesma interface que o TDA Fila, mas semântica diferente. Mais uma vez, a interface é a seguinte:

__init__ Inicializar uma nova fila vazia.

insert Adicionar um novo item à fila.

remove Remover e retornar um item da fila. O item retornado é aquele que tiver maior prioridade.

isEmpty Checar se a fila está vazia.

A diferença semântica é que o item removido da fila não é necessariamente o que foi incluído primeiro e, sim, o que tem maior prioridade. Que prioridades são essas e como elas se comparam umas com as outras não é especificado pela implementação Fila por Prioridade. Isso depende de quais itens estão na fila.

Por exemplo, se os itens da fila tiverem nome, podemos escolhê-los por ordem alfabética. Se for a pontuação de um jogo de boliche, podemos ir da maior para a menor, mas se for pontuação de golfe, teríamos que ir da menor para a maior. Se é possível comparar os itens da fila, é possível achar e remover o que tem maior prioridade. Essa implementação da Fila por Prioridade tem como atributo uma lista Python chamada `items`, que contém os itens da fila.

```
class PriorityQueue:
  def __init__(self):
    self.items = []

  def isEmpty(self):
    return self.items == []

  def insert(self, item):
    self.items.append(item)
```

O método de inicialização, `isEmpty`, e `insert` são apenas uma fachada para operações básicas de lista. O único método interessante é `remove`:

```
class PriorityQueue:
  # ...
  def remove(self):
    maxi = 0
    for i in range(1,len(self.items)):
      if self.items[i] > self.items[maxi]:
        maxi = i
    item = self.items[maxi]
    self.items[maxi:maxi+1] = []
    return item
```

No início de cada iteração, `maxi` armazena o índice do maior item (a prioridade mais alta de todas) que vimos *até agora*. A cada volta do laço, o programa compara o `i`-ésimo item ao campeão. Se o novo item for maior, `maxi` recebe o valor de `i`.

Quando o comando `for` se completa, `maxi` é o índice do maior item. Esse item é removido da lista e retornado.

Vamos testar a implementação:

```
>>> q = PriorityQueue()
>>> q.insert(11)
>>> q.insert(12)
>>> q.insert(14)
>>> q.insert(13)
>>> while not q.isEmpty(): print q.remove()
14
13
12
11
```

Se a fila contém números ou strings simples, eles são removidas em ordem numérica decrescente ou alfabética invertida (de Z até A). Pyhton consegue achar o maior inteiro ou string porque consegue compará-los usando os operadores de comparação nativos.

Se a fila contém objetos de outro tipo, os objetos têm que prover um método `__cmp__`. Quando `remove` usa o operador > para comparar dois itens, o método `__cmp__` de um dos itens é invocado, recebendo o segundo item como argumento. Desde que o método `__cmp__` funcione de forma consistente, a Fila por Prioridade vai funcionar.

21.6 19.6 A classe `Golfer`

Como exemplo de um objeto com uma definição não-usual de prioridade, vamos implementar uma classe chamada `Golfer` (golfista), que mantém registro dos nomes e da pontuação de golfistas. Como sempre, começamos definindo `__init__` e `__str__`:

```
class Golfer:
  def __init__(self, name, score):
    self.name = name
    self.score= score

  def __str__(self):
    return "%-16s: %d" % (self.name, self.score)
```

O método `__str__` usa o operador de formato para colocar nomes e pontuações em colunas arrumadas.

Em seguida, definimos uma versão de `__cmp__`, ma qual a pontuação mais baixa fica com prioridade máxima. Como sempre, `__cmp__` retorna 1 se `self` é "maior que" `other`, -1 se `self` é "menor que" `other`, e 0 se eles são iguais.

```
class Golfer:
  #...
  def __cmp__(self, other):
    if self.score < other.score: return  1   # less is more
    if self.score > other.score: return -1
    return 0
```

Agora estamos prontos para testar a fila por prioridade com a classe `Golfer`:

```
>>> tiger = Golfer("Tiger Woods",    61)
>>> phil  = Golfer("Phil Mickelson", 72)
>>> hal   = Golfer("Hal Sutton",     69)
>>>
>>> pq = PriorityQueue()
>>> pq.insert(tiger)
>>> pq.insert(phil)
```

```
>>> pq.insert(hal)
>>> while not pq.isEmpty(): print pq.remove()
Tiger Woods    : 61
Hal Sutton     : 69
Phil Mickelson : 72
```

Como exercício, escreva uma implementação do TDA Fila por Prioridade usando uma lista encadeada. Mantenha a lista em ordem para que a remoção seja uma operação de tempo constante. Compare a performance dessa implementação com a implementação usando uma lista nativa do Python.

21.7 19.7 Glossário

fila (*queue*) Conjunto de objetos ordenados esperando algum tipo de serviço.

Fila (*Queue*) TAD (Tipo Abstrato de Dado) que realiza operações comuns de acontecerem em uma fila.

política de enfileiramento (*queueing policy*) As regras que determinam qual membro de uma fila é o próximo a ser removido.

FIFO "First In, First Out," (primeiro a entrar, primeiro a sair) política de enfileiramento em que o primeiro membro a chegar é o primeiro a ser removido.

fila por prioridade (*priority queue*) Política de enfileiramento em que cada membro tem uma prioridade, determinada por fatores externos. O membro com a maior prioridade é o primeiro a ser removido.

Fila por Prioridade (*Priority Queue*) TAD que define as operações comuns de acontecerem em uma fila por prioridade.

fila encadeada (*linked queue*) Implementação de uma fila usando uma lista encadeada.

tempo constante (*constant time*) Operação cujo tempo de execução não depende do tamanho da estrutura de dados.

tempo linear (*linear time*) Operação cujo tempo de execução é uma função linear do tamanho da estrutura de dados.

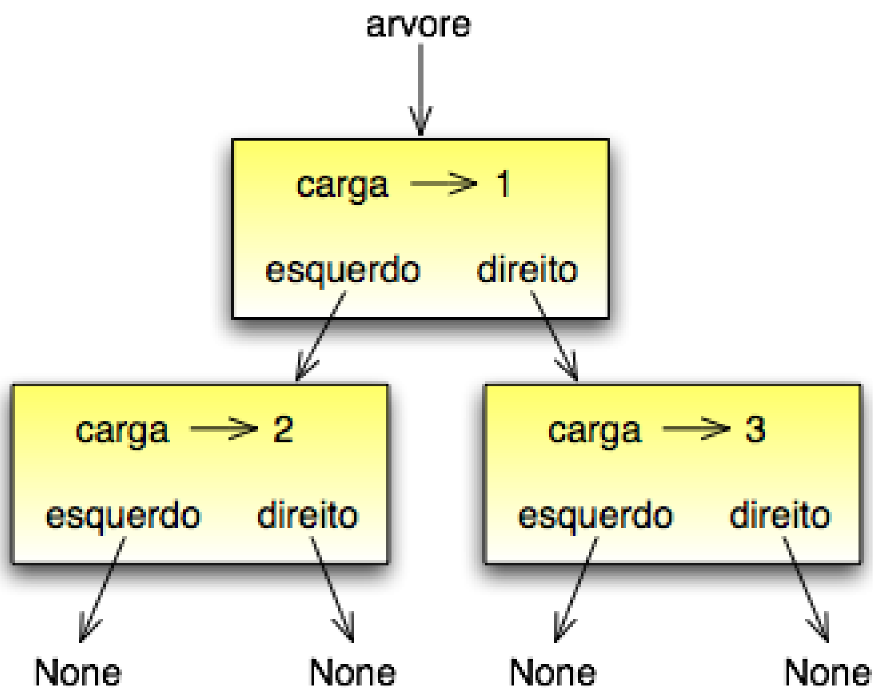
Capítulo 20: Árvores

Tópicos

Nota: Veja a discussão sobre o vocabulário usado no fim da página.

Como listas ligadas, árvores são constituídas de células. Uma espécie comum de árvores é a **árvore binária**, em que cada célula contém referências a duas outras células (possivelmente nulas). Tais referências são chamadas de subárvore esquerda e direita. Como as células de listas ligadas, as células de árvores também contém uma carga. Um diagrama de estados para uma árvore pode aparecer assim:

Figura 1



Para evitar a sobrecarga da figura, nós frequentemente omitimos os `Nones`.

O topo da árvore (a célula à qual o apontador `tree` se refere) é chamada de **raiz**. Seguindo a metáfora das árvores, as outras células são chamadas de galhos e as células nas pontas contendo as referências vazia são chamadas de **folhas**. Pode parecer estranho que desenhamos a figura com a raiz em cima e as folhas em baixo, mas isto nem será a coisa mais estranha.

Para piorar as coisas, cientistas da computação misturam outra metáfora além da metáfora biológica - a árvore genealógica. Uma célula superior pode ser chamada de **pai** e as células a que ela se refere são chamadas de seus **filhos**. Células com o mesmo pai são chamadas de **irmãos**.

Finalmente, existe também o vocabulário geométrico para falar de árvores. Já mencionamos esquerda e direita, mas existem também as direções "para cima" (na direção da raiz) e "para baixo" (na direção dos filhos/folhas). Ainda nesta terminologia, todas as células situadas à mesma distância da raiz constituem um **nível** da árvore.

Provavelmente não precisamos de três metáforas para falar de árvores, mas aí elas estão.

Como listas ligadas, árvores são estruturas de dados recursivas já que elas são definidas recursivamente:

> Uma árvore é
>
> - a árvore vazia, representada por `None`, ou
> - uma célula que contém uma referência a um objeto (a carga da célula) e duas referências a árvores.

22.1 20.1 Construindo árvores

O processo de montar uma árvore é similar ao processo de montar uma lista ligada. cada invocação do construtor cria uma célula.

```
class Tree :
  def __init__(self, cargo, left=None, right=None) :
    self.cargo = cargo
    self.left  = left
    self.right = right

  def __str__(self) :
    return str(self.cargo)
```

A `carga` pode ser de qualquer tipo, mas os parâmetros `left` e `right` devem ser células. `left` e `right` são opcionais; o valor default é `None`.

Para imprimir uma célula, imprimimos apenas a sua carga.

Uma forma de construir uma árvore é de baixo para cima. Aloque os filhos primeiro:

```
left = Tree(2)
right = Tree(3)
```

Em seguida crie a célula pai e ligue ela a seus filhos:

```
tree = Tree(1, left, right);
```

Podemos escrever este código mais concisamente encaixando as invocações do construtor:

```
>>> tree = Tree(1, Tree(2), Tree(3))
```

De qualquer forma, o resultado é a árvore que apareceu no início do capítulo.

22.2 20.2 Percorrendo árvores

Cada vez que Você vê uma nova estrutura de dados, sua primeira pergunta deveria ser "Como eu percorro esta estrutura?" A forma mais natural de percorrer uma árvore é fazer o percurso recursivamente. Por exemplo, se a árvore contém inteiros na carga, a função abaixo retorna a soma das cargas:

```
def total(tree) :
  if tree == None : return 0
  return total(tree.left) + total(tree.right) + tree.cargo
```

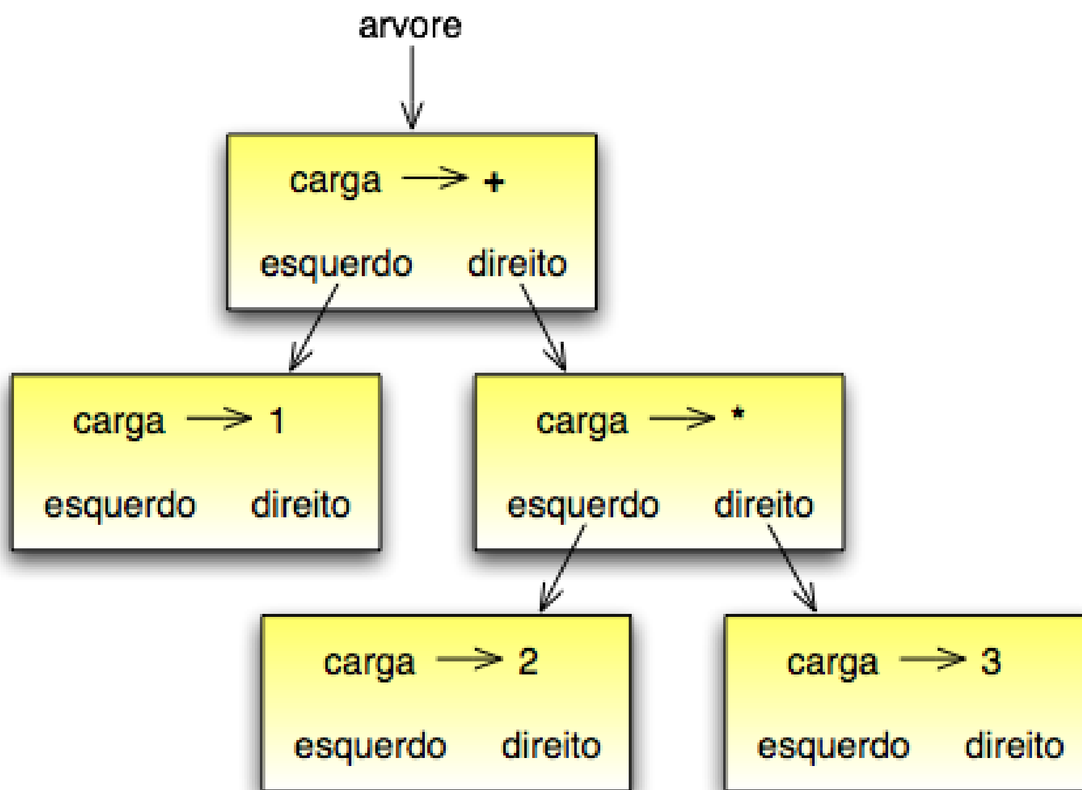
O caso base é a árvore vazia, que não contém nenhuma carga, logo a soma das cargas é 0. O passo recursivo faz duas chamadas recursivas para achar a soma das cargas das subárvores dos filhos. Ao finalizar a chamada recursiva, adicionamos a carga do pai e devolvemos o valor total.

22.3 20.3 Árvores de expressões

Uma árvore é uma forma natural para representar a estrutura de uma expressão. Ao contrário de outras notações, a árvore pode representar a computação de forma não ambígua. Por exemplo, a expressão infixa `1 + 2 * 3` é ambígua, a menos que saibamos que a multiplicação é feita antes da adição.

A árvore de expressão seguinte representa a mesma computação:

Figura 2



As células de uma árvore de expressão podem ser operandos como `1` e `2` ou operações como `+` e `*`. As células contendo operandos são folhas; aquelas contendo operações devem ter referências aos seus operandos. (Todos os nossos operandos são **binários**, significando que eles tem exatamente dois operandos.)

Podemos construir árvores assim:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
```

Examinando a figura, não há dúvida quanto à ordem das operações; a multiplicação é feita primeiro para calcular o segundo operando da adição.

Árvores de expressão tem muitos usos. O exemplo neste capítulo usa árvores para traduzir expressões para as notações pósfixa, prefixa e infixa. Árvores similares são usadas em compiladores para analisar sintaticamente, otimizar e traduzir programas.

22.4 20.4 Percurso de árvores

Podemos percorrer uma árvore de expressão e imprimir o seu conteúdo como segue:

```
def printTree(tree) :
  if tree == None : return
  print tree.cargo,
  printTree(tree.left)
  printTree(tree.right)
```

Em outras palavras, para imprimir uma árvore, imprima primeiro o conteúdo da raiz, em seguida imprima toda a subárvore esquerda e finalmente imprima toda a subárvore direita. Esta forma de percorrer uma árvore é chamada de **préordem**, porque o conteúdo da raiz aparece *antes* dos conteúdos dos filhos. Para o exemplo anterior, a saída é:

```
>>> tree = Tree('+', Tree(1), Tree('*', Tree(2), Tree(3)))
>>> printTree(tree)
+ 1 * 2 3
```

Esta notação é diferente tanto da notação pósfixa quanto da infixa; é uma notação chamada de **prefixa**, em que os operadores aparecem antes dos seus operandos.

Você pode suspeitar que se Você percorre a árvore numa ordem diferente, Você produzirá expressões numa notação diferente. Por exemplo, se Você imprime subárvores primeiro e depois a raiz, Você terá:

```
def printTreePostorder(tree) :
  if tree == None : return
  printTreePostorder(tree.left)
  printTreePostorder(tree.right)
  print tree.cargo,
```

O resultado, `1 2 3 * +`, está na notação pósfixa! Esta ordem de percurso é chamada de pósordem.

Finalmente, para percorrer uma árvore em **inordem**, Você imprime a subárvore esquerda, depois a raiz e depois a subárvore direita:

```
def printTreeInorder(tree) :
  if tree == None : return
  printTreeInorder(tree.left)
  print tree.cargo,
  printTreeInorder(tree.right)
```

O resultado é `1 + 2 * 3`, que é a expressão na notação infixa.

Para sermos justos, devemos lembrar que acabamos de omitir uma complicação importante. algumas vezes quando escrevemos expressões na notação infixa devemos usar parêntesis para prescrever a ordem das operações. Ou seja, um percurso em inordem não é suficiente para gerar a expressão infixa.

Ainda assim, com alguns aperfeiçoamentos, a árvore de expressão e os três modos recursivos de percurso resultam em algoritmos para transformar expressões de uma notação para outra.

> *Como um exercício, modifique 'printTreeInorder' de modo que ele coloque parêntesis em volta de cada operador e par de operandos. A saída é correta e não ambígua? Os parêntesis são sempre necessários?*

Se percorrermos uma árvore em inordem e acompanharmos em qual nível na árvore estamos, podemos gerar uma representação gráfica da árvore:

```
def printTreeIndented(tree, level=0) :
  if tree == None : return
  printTreeIndented(tree.right, level+1)
  print '  '*level + str(tree.cargo)
  printTreeIndented(tree.left, level+1)
```

O parâmetro `level` registra aonde estamos na árvore. Por 'default', o nível inicialmente é zero. A cada chamada recursiva repassamos `level+1` porque o nível do filho é sempre um a mais do que o nível do pai. Cada item é indentado dois espaços por nível. Para o nosso exemplo obtemos:

```
>>> printTreeIndented(tree)
    3
  *
    2
```

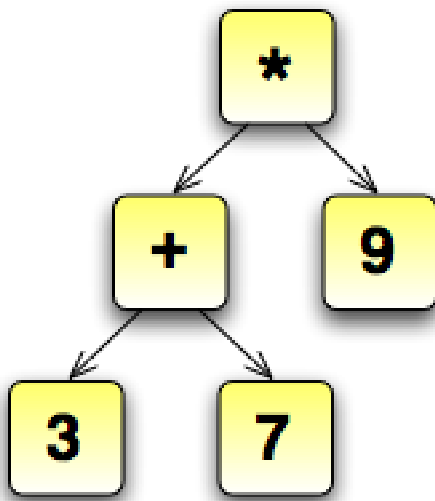
```
+
  1
```

Se Você deitar a saída acima Você enxerga uma versão simplificada da figura original.

22.5 20.5 Construindo uma árvore de expressão

Nesta seção analisamos expressões infixas e construímos as árvores de expressã correspondentes. Por exemplo, para a expressão `(3+7)*9` resultará a seguinte árvore:

Figura 3



Note que simplificamos o diagrama omitindo os nomes dos campos.

O analisador que escreveremos aceitará expressões que incluam números, parêntesis e as operações `+` e `*`. Vamos supor que a cadeia de entrada já foi tokenizada numa lista do Python. A lista de tokens para a expressão `(3+7)*9` é:

```
['(', 3, '+', 7, ')', '*', 9, 'end']
```

O token final `end` é prático para prevenir que o analisador tente buscar mais dados após o término da lista.

> *A título de um exercício, escreva uma função que recebe uma expressão na forma de uma cadeia e devolve a lista de tokens.*

A primeira função que escreveremos é `getToken` que recebe como parâmetros uma lista de tokens e um token esperado. Ela compara o token esperado com o o primeiro token da lista: se eles batem a função remove o token da lista e devolve um valor verdadeiro, caso contrário a função devolve um valor falso:

```
def getToken(tokenList, expected) :
  if tokenList[0] == expected :
    tokenList[0:1] = []   # remove the token
    return 1
  else :
    return 0
```

Já que `tokenList` refere a um objeto mutável, as alterações feitas aqui são visíveis para qualquer outra variável que se refira ao mesmo objeto.

A próxima função, `getNumber`, trata de operandos. Se o primeiro token na `tokenList` for um número então `getNumber` o remove da lista e devolve uma célula folha contendo o número; caso contrário ele devolve `None`.

```
def getNumber(tokenList) :
  x = tokenList[0]
  if type(x) != type(0) : return None
  del tokenList[0]
  return Tree(x, None, None)
```

Antes de continuar, convém testar `getNumber` isoladamente. Atribuímos uma lista de números a `tokenList`, extraímos o primeiro, imprimimos o resultado e imprimimos o que resta na lista de tokens:

```
>>> tokenList = [9, 11, 'end']
>>> x = getNumber(tokenList)
>>> printTreePostorder(x)
9
>>> print tokenList
[11, 'end']
```

Em seguida precisaremos da função `getProduct`, que constrói uma árvore de expressão para produtos. Os dois operandos de um produto simples são números, como em `3 * 7`.

Segue uma versão de `getProduct` que trata de produtos simples.

```
def getProduct(tokenList) :
  a = getNumber(tokenList)
  if getToken(tokenList, '*') :
    b = getNumber(tokenList)
    return Tree('*', a, b)
  else :
    return a
```

Supondo que a chamada de `getNumber` seja bem sucedida e devolva uma árvore de uma só célula atribuímos o primeiro operando a *à*‘. Se o próximo caractere for `*`, vamos buscar o segundo número e construir a árvore com `a`, `b` e o operador.

Se o caractere seguinte for qualquer outra coisa, então simplesmente devolvemos uma célula folha com `a`. Seguem dois exemplos:

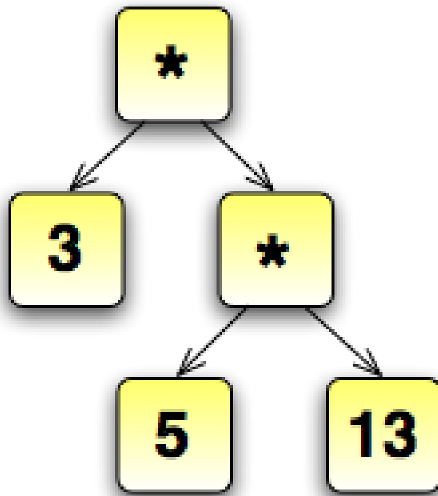
```
>>> tokenList = [9, '*', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9 11 *


>>> tokenList = [9, '+', 11, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
9
```

O segundo exemplo sugere que nós consideramos um operando unitário como uma espécie de produto. Esta definição de “produto” talvez não seja intuitiva, mas ela será útil.

Agora tratamos produtos compostos, como `3 * 5 * 13`. Encaramos esta expressão como um produto de produtos, mais precisamente como `3 * (5 * 13)`. A árvore resultante é:

Figura 4



Com uma pequena alteração em `getProduct`, podemos acomodar produtos arbitrariamente longos:

```
def getProduct(tokenList) :
  a = getNumber(tokenList)
  if getToken(tokenList, '*') :
    b = getProduct(tokenList)        # this line changed
    return Tree('*', a, b)
  else :
    return a
```

Em outras palavras, um produto pode ser um singleton ou uma árvore com `*` na raiz, que tem um número como filho esquerdo e um produto como filho direito. Este tipo de definição recursiva devia começar a ficar familiar.

Testemos a nova versão com um produto composto:

```
>>> tokenList = [2, '*', 3, '*', 5 , '*', 7, 'end']
>>> tree = getProduct(tokenList)
>>> printTreePostorder(tree)
2 3 5 7 * * *
```

A seguir adicionamos o tratamento de somas. De novo, usamos uma definição de "soma" que é ligeiramente não intuitiva. Para nós, uma soma pode ser uma árvore com + na raiz, que tem um produto como filho esquerdo e uma soma como filho direito. Ou, uma soma pode ser simplesmente um produto.

Se Você está disposto a brincar com esta definição, ela tem uma propriedade interessante: podemos representar qualquer expressão (sem parêntesis) como uma soma de produtos. Esta propriedade é a base do nosso algoritmo de análise sintática.

`getSum` tenta construir a árvore com um produto à esquerda e uma soma à direita. Mas, se ele não encontra uma `+`, ele simplesmente constrói um produto.

```
def getSum(tokenList) :
  a = getProduct(tokenList)
  if getToken(tokenList, '+') :
    b = getSum(tokenList)
    return Tree('+', a, b)
```

```
  else :
    return a
```

Vamos testar o algoritmo com `9 * 11 + 5 * 7`:

```
>>> tokenList = [9, '*', 11, '+', 5, '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 * 5 7 * +
```

Quase terminamos, mas ainda temos que tratar dos parêntesis. Em qualquer lugar numa expressão onde podemos ter um número, podemos também ter uma soma inteira envolvida entre parêntesis. Precisamos, apenas, modificar `getNumber` para que ela possa tratar de **subexpressões**:

```
def getNumber(tokenList) :
  if getToken(tokenList, '(') :
    x = getSum(tokenList)       # get subexpression
    getToken(tokenList, ')')     # eat the closing parenthesis
    return x
  else :
    x = tokenList[0]
    if type(x) != type(0) : return None
    tokenList[0:1] = []    # remove the token
    return Tree(x, None, None)     # return a leaf with the number
```

Testemos este código com `9 * (11 + 5) * 7`:

```
>>> tokenList = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']
>>> tree = getSum(tokenList)
>>> printTreePostorder(tree)
9 11 5 + 7 * *
```

O analisador tratou os parêntesis corretamente; a adição é feita antes da multiplicação.

Na versão final do programa, seria uma boa idéia dar a `getNumber` um nome mais descritivo do seu novo papel.

22.6 20.6 Manipulando erros

Ao longo do analisador sintático tínhamos suposto que as expressões (de entrada) são bem formadas. Por exemplo, quando atingimos o fim de uma subexpressão, supomos que o próximo caractere é um facha parêntesis. Caso haja um erro e o próximo caractere seja algo diferente, devemos tratar disto.

```
def getNumber(tokenList) :
  if getToken(tokenList, '(') :
    x = getSum(tokenList)
    if not getToken(tokenList, ')'):
      raise 'BadExpressionError', 'missing parenthesis'
    return x
  else :
    # the rest of the function omitted
```

O comando `raise` cria uma exceção; neste caso criamos um novo tipo de exceção, chamada de `BadExpressionError`. Se a função que chamou `getNumber`, ou uma das outras funções no traceback, manipular a exceção, então o programa pode continuar. caso contrário Python vai imprimir uma mensagem de erro e terminará o processamento em seguida.

> *A título de exercício, encontre outros locais nas funções criadas onde erros possam ocorrer e adiciona comandos ``raise`` apropriados. Teste seu código com expressões mal formadas.*

22.7 20.7 A árvore dos animais

Nesta seção, desenvolvemos um pequeno programa que usa uma árvore para representar uma base de conhecimento.

O programa interage com o usuário para criar uma árvore de perguntas e de nomes de animais. Segue uma amostra da funcionalidade:

```
Are you thinking of an animal? y
Is it a bird? n
What is the animals name? dog
What question would distinguish a dog from a bird? Can it fly
If the animal were dog the answer would be? n

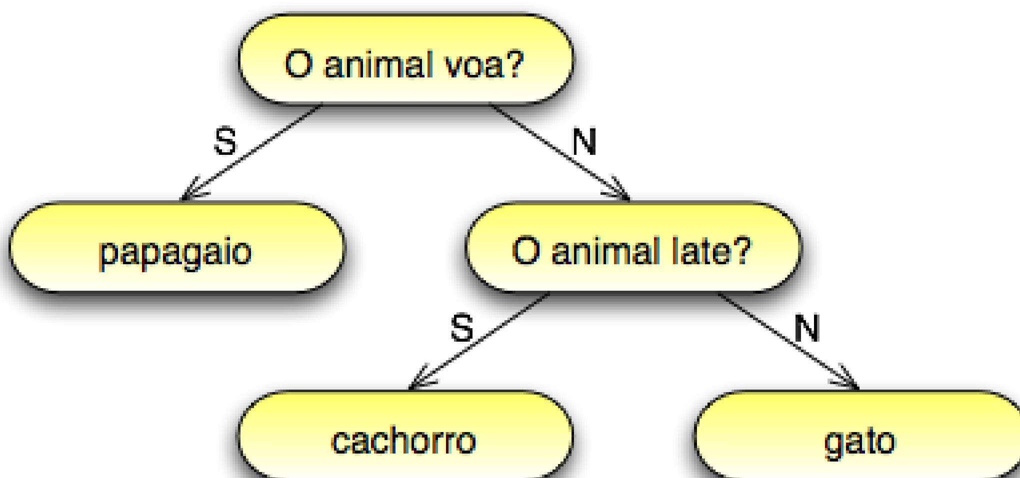
Are you thinking of an animal? y
Can it fly? n
Is it a dog? n
What is the animals name? cat
What question would distinguish a cat from a dog? Does it bark
If the animal were cat the answer would be? n

Are you thinking of an animal? y
Can it fly? n
Does it bark? y
Is it a dog? y
I rule!

Are you thinking of an animal? n
```

Aqui está a árvore que este diálogo constrói:

Figura 5



No começo de cada rodada, o programa parte do topo da árvore e faz a primeira pergunta. Dependendo da resposta, ele segue pelo filho esquerdo ou direito e continua até chegar numa folha. Neste ponto ele arrisca um palpite. Se o palpite não for correto, ele pergunta ao usuário o nome de um novo animal e uma pergunta que distingue o palpite errado do novo animal. A seguir, adiciona uma célula à árvore contendo a nova pergunta e o novo animal.

Aqui está o código:

```
def animal() :
   # start with a singleton
   root = Tree("bird")

   # loop until the user quits
   while 1 :
     print
     if not yes("Are you thinking of an animal? ") : break

     # walk the tree
     tree = root
     while tree.getLeft() != None :
       prompt = tree.getCargo() + "? "
       if yes(prompt):
         tree = tree.getRight()
       else:
         tree = tree.getLeft()

     # make a guess
     guess = tree.getCargo()
     prompt = "Is it a " + guess + "? "
     if yes(prompt) :
       print "I rule!"
       continue

     # get new information
     prompt   = "What is the animal\'s name? "
     animal   = raw_input(prompt)
     prompt   = "What question would distinguish a %s from a %s? "
     question = raw_input(prompt % (animal,guess))

     # add new information to the tree
     tree.setCargo(question)
     prompt = "If the animal were %s the answer would be? "
     if yes(prompt % animal) :
       tree.setLeft(Tree(guess))
       tree.setRight(Tree(animal))
     else :
       tree.setLeft(Tree(animal))
       tree.setRight(Tree(guess))
```

A função `yes` é um auxiliar; ele imprime um prompt e em seguida solicita do usuário uma entrada. Se a resposta começar com `y` ou `Y`, a função devolve um valor verdadeiro:

```
def yes(ques) :
  from string import lower
  ans = lower(raw_input(ques))
  return (ans[0:1] == 'y')
```

A condição do laço externo é `1`', que significa que ele continuará até a execução de um comando `break`, caso o usuário não pense num animal.

O laço `while` interno caminha na árvore de cima para baixo, guiado pelas respostas do usuário.

Quando uma nova célula é adicionada à árvore, a nova pergunta substitui a carga e os dois filhos são o novo animal e a carga original.

Uma falha do programa é que ao sair ele esquece tudo que lhe foi cuidadosamente ensinado!

A título de exercício, pense de várias jeitos para salvar a árvore do conhecimento acumulado num arquivo. Implemente aquele que Você pensa ser o mais fácil.

22.8 20.8 Glossário

árvore binária (*binary tree*) Uma árvore em que cada célula tem zero, um ou dois descendentes.

raiz (*root*) A célula mais alta de uma árvore, a (única) célula de uma árvore que não tem pai.

folha (*leaf*) Uma célula mais baixa numa árvore; uma célula que não tem descendentes.

pai (*parent*) A célula que aponta para uma célula dada.

filho (*child*) Uma célula apontada por uma célula dada.

irmãos (*siebling*) Células que tem o mesmo pai.

nível (*level*) Um conjunto de células equidistantes da raiz.

operador binário (*binary operator*) Um operador sobre dois operandos.

subexpressão (*subexpression*) Uma expressão entre parêntesis que se comporta como um operando simples numa expressão maior.

pré-ordem (*preorder*) Uma forma de percorrer uma árvore visitando cada célula antes dos seus filhos.

notação prefixa (*prefix notation*) Uma forma de escrever uma expressão matemática em que cada operador aparece antes dos seus operandos.

pós-ordem (*postorder*) Uma forma de percorrer uma árvore visitando os filhos de cada célula antes da própria célula.

in-ordem (*inorder*) Uma forma de percorrer uma árvore visitando a subárvore esquerda, seguida da raiz e finalmente da subárvore direita.

CAPÍTULO 23

Apêndice A: Depuração

Tópicos

Diferentes tipos de erros podem acontecer em um programa, e é útil distinguir entre eles para os localizar mais rapidamente:

- Erros de sintaxe são produzidos por Python quando o interpretador está traduzindo o código fonte em *bytecode*. Estes erros geralmente indicam que existe algo errado com a sintaxe do programa. Exemplo: Omitir o sinal de dois pontos (`:`) no final de uma declaração `def` produz a mensagem um tanto redundante `SintaxError: invalid sintax` (Erro de sintaxe: sintaxe inválida).
- Erros de tempo de execução são produzidos se algo de errado acontece enquanto o programa está em execução. A maioria das mensagens de tempo de execução incluem informação sobre onde o erro ocorreu e que função estava em execução. Exemplo: Uma recursão infinita eventualmente causa um erro em tempo de execução identificado como `maximum recursion depth exceeded` (Excedida a máxima profundidade de recursão).
- Erros de semântica, chamado por alguns autores de erros de lógica, são problemas com um programa que compila e executa mas não tem o resultado esperado. Exemplo: Uma expressão pode não ser avaliada da forma que você espera, produzindo um resultado inesperado.

O primeiro passo na depuração é descobrir com que tipo de erro você está lidando. Embora as seções a seguir estejam organizadas por tipo de erro, algumas técnicas são aplicáveis em mais de uma situação.

23.1 A.1 Erros de sintaxe

Erros de sintaxe são geralmente fáceis de corrigir, bastando apenas que você descubra onde eles estão. Infelizmente, as mensagens de erro geralmente ajudam pouco. As mensagens mais comuns são: `SyntaxError: invalid syntax` (Erro de sintaxe: sintaxe inválida) e `SyntaxError: invalid token` (Erro de sintaxe: objeto inválido), nenhuma das duas é muito informativa.

Por outro lado, a mensagem diz a você onde, no código do programa, ocorreu o problema. Na verdade, ela diz a você onde o Python encontrou o problema, que não é necessariamente onde o erro está. Às vezes o erro é anterior à localização da mensagem de erro, geralmente na linha precedente.

Se você está construindo o programa incrementalmente, você terá uma boa ideia sobre onde estará o erro. Estará na última linha adicionada.

Se você está copiando código de um livro, comece comparando seu código ao código do livro de forma muito cuidadosa. Verifique cada caracter. Ao mesmo tempo, lembre-se que o livro pode estar errado, então você pode perfeitamente encontrar um erro de sintaxe em um livro.

Aqui estão algumas maneiras de evitar os erros de sintaxe mais comuns:

1. Certifique-se que você não está utilizando uma palavra reservada de Python para um nome de variável.
2. Verifique a existência do sinal de dois pontos no final do cabeçalho de cada declaração composta, incluindo as declarações `for`, `while`, `if`, e `def`.
3. Verifique se a endentação está consistente. Você pode endentar com espaços ou com tabulações, mas é melhor não misturá-los. Cada nível deve ser aninhado com a mesma quantidade.
4. Assegure-se de que cada *string* no código tenha as aspas correspondentes.
5. Se você tem *strings* de multilinhas criadas usando três aspas (simples ou duplas), assegure-se de que você terminou a *string* apropriadamente. Uma *string* terminada de forma inapropriada ou não terminada pode gerar um erro de `invalid token` (objeto inválido) no final do seu programa, ou ele pode tratar a parte seguinte do programa como uma *string* até chegar à próxima *string*. No segundo caso, pode ser que o interpretador nem mesmo produza uma mensagem de erro!
6. Um conjunto de parênteses, colchetes ou chaves não fechados corretamente faz com que o Python continue com a próxima linha como parte da declaração anterior. Geralmente, um erro ocorre quase imediatamente na linha seguinte.
7. Verique o clássico = ao invés de == dentro de uma condicional.

Se nada funcionar, vá para a próxima seção.

23.1.1 A.1.1 Eu não consigo fazer meu programa executar não importa o que eu faça

Se o compilador diz que existe um erro e você não o vê, isto pode ser por que você e o compilador não estão olhando para o mesmo código. Verifique seu ambiente para se assegurar que o programa que você está editando é o mesmo que o Python está tentando executar. Se você não está certo, tente colocar um erro de sintaxe óbvio e deliberado no início do programa. Agora execute (ou importe) o programa novamente. Se o compilador não encontrar o novo erro, provavelmente existe algo de errado com a maneira como o ambiente está configurado.

Se isto acontecer, uma abordagem é iniciar novamente com um novo programa como "Hello World!", e se assegurar de que você consegue colocar um programa conhecido para executar. Então gradualmente adicione as partes do novo programa ao programa que está funcionando.

23.2 A.2 Erros de tempo de execução

Uma vez que seu programa está sintaticamente correto, o interpretador do Python pode importá-lo e começar a executá-lo. O que poderia dar errado?

23.2.1 A.2.1 Meu programa não faz absolutamente nada

Este é o problema mais comum quando um arquivo consiste de funções e classes mas não chama nada realmente para iniciar a execução. Isto pode ser intencional se você planeja apenas importar este módulo para fornecer classes e funções.

Se isto não é intencional, assegure-se de que você está chamando uma função para iniciar a execução ou execute uma "manualmente" do *prompt* interativo. Veja também a seção "Fluxo de Execução" abaixo.

23.2.2 A.2.2 Meu programa trava

Se um programa para e parece não estar fazendo nada, dizemos que ele "travou". Geralmente isto significa que ele foi pego num laço infinito ou numa recursão infinita.

- Se existe um laço em particular aonde você suspeita estar o problema, adicione uma declaração `print` imediatamente antes do laço que diga "entrando no laço" e uma outra imediatamente depois que diga "saindo do laço". Execute o programa. Se você receber a primeira mensagem e não a segunda, você tem um laço infinito. Vá para a seção "Laço Infinito" abaixo.
- Na maioria das vezes, uma recursão infinita fará com que o programa execute por um tempo e então ele produzirá um erro do tipo `Runtime Error: Maximum recursion depth exceeded` (Erro de tempo de execução: Excedida a profundidade máxima de recursão). Se isto acontecer, vá para a seção "Recursão Infinita" abaixo. Se você não está recebendo este erro, mas suspeita que há um problema com um método ou função recursiva, você ainda pode utilizar as técnicas da seção "Recursão Infinita".
- Se nenhum destes passos funcionar, comecea testar outros laços ou métodos ou funções recursivas.
- Se isso não funcionar, então é possível que você não entenda o fluxo de execução do seu programa. Vá para a seção "Fluxo de Execução" abaixo.

Laço Infinito

Se você acha que tem um laço infinito e desconfia de qual seja laço causador do problema, adicione uma declaração `print` no final do laço que imprima os valores das variáveis na condição e o valor da condição.

Por exemplo:

```
>>> while x > 0 and y < 0:
        # faz algo para x
        # faz algo para y
    print "x: ", x
    print "y: ", y
    print "condição: ", (x > 0 and y < 0)
```

Agora, quando você executar o programa, serão exibidas três linhas de saída para cada execução do laço. Na última execução do laço, a condição deveria ser `False` (falso). Se o laço continuar, você terá condições de ver os valores de `x` e `y`, podendo, assim, descobrir o porquê das variáveis não serem atualizadas corretamente.

Recursão Infinita

Na maioria das vezes, uma recursão infinita fará com que um programa execute por um determinado tempo e então produza um erro de `Maximum recursion depth exceeded` (Excedida a profundidade máxima de recursão).

Se você suspeita que uma função ou método está causando uma recursão infinita, comece verificando para se assegurar que exista um caso básico. Em outras palavras, deve existir alguma condição que faça com que a função ou método finalize sem fazer uma chamada recursiva. Se não existe tal condição, você precisa repensar o algoritmo e identificar um caso base.

Se existe um caso base mas o programa parece não estar alcançando-o, adicione uma declaração `print` no início da função ou método que imprime o(s) parâmetro(s). Agora, quando você executar o programa, você verá umas poucas linhas de saída todas as vezes que a função ou método for executado, e você verá o(s) parâmetro(s). Se o(s) parâmetro(s) não está(ão) se movendo em direção ao caso base, você terá ideias de por que não, e poderá então corrigir o problema.

Fluxo de Execução

Se você não está certo de como o seu programa está fluindo, adicione declarações `print` ao início de cada função com uma mensagem semelhante a "entrando na função `foo`", onde `foo` é o nome da função.

Agora quando você executar o programa, será exibido um rastro de cada função à medida em que ela é invocada.

23.2.3 A.2.3 Quando eu executo o programa, recebo uma exceção

Se algo vai errado durante a execução, o Python exibe uma mensagem que inclui o nome da exceção, a linha do código do programa onde o problema ocorre, e dados para investigação do erro, onde é descrita a pilha de execução de funções e métodos.

Tais dados identificam a função que está sendo executada no momento, e então a função que a invocou, e então a função que invocou *aquela*, e assim por diante (a pilha de execução de funções). Em outras palavras, ele traça o caminho das invocações da função que te levou até onde você está. Ele também inclui o número da linha no respectivo arquivo onde cada uma dessas chamadas ocorre.

O primeiro passo é examinar o lugar no programa onde ocorreu o erro e tentar descobrir o que aconteceu. Esses são alguns dos erros de tempo de execução mais comuns:

NameError (Erro de Nome): Você está tentando utilizar uma variável que não existe no ambiente atual. Lembre-se que variáveis locais são locais. Você não pode referenciá-la fora da função onde ela foi definida.

TypeError (Erro de Tipo): Existem várias causas possíveis:

- Você está tentando utilizar um valor de forma imprópria. Exemplo: indexando uma *string*, lista ou tupla com alguma coisa que não é um inteiro.
- Há uma incompatibilidade entre os itens em um formato de *string* e os itens passados para conversão. Isto pode acontecer se o número de itens não for igual ou se uma conversão inválida é chamada. Por exemplo: Passar uma *string* para a formatação de conversão `%f`.
- Você está passando o número errado de argumentos para uma função ou método. Para métodos, verifique sua definição e confira se o primeiro parâmetro chama-se `self`. Então verifique a chamada ao método; certifque-se de que você está chamando o método em um objeto com o tipo correto e fornecendo os argumentos adequados.

KeyError (Erro de Chave): Você está tentando acessar um elemento de um dicionário utilizando um valor de chave que o dicionário não contém.

AttributeError (Erro de Atributo): Você está tentando acessar um atributo ou método que não existe em um objeto.

IndexError (Erro de Índice): O índice que você está usando para acessar uma lista, *string* ou tupla não existe no objeto, ou seja, é maior que seu comprimento menos um. Imediatamente antes do ponto do erro, adicione uma delaração *print* para mostrar o valor do índice e o comprimento do *objeto*. O *objeto* é do tamanho correto? O índice está com o valor adequado?

23.2.4 A.2.4 Eu adicionei tantas declarações *print* que a saída do programa ficou bagunçada

Um dos problemas com o uso de declarações *print* para a depuração é que a saída pode ficar confusa, dificultando a depuração, ao invés de facilitar. Há duas coisas que podem ser feitas: simplificar a saída ou simplificar o programa.

Para simplificar a saída, você pode remover ou comentar as declarações *print* que não estão ajudando, ou combiná-las, ou, ainda, formatar a saída para facilitar o entendimento.

Para simplificar o programa, existem várias coisas que você pode fazer: Primeiro, reduza o problema no qual o programa está trabalhando. Por exemplo, se você está ordenando um *array*, ordene um * array* **pequeno**. Se o programa recebe entrada do usuário, dê a ele a entrada mais simples capaz de causar o problema.

Segundo, limpe o programa. Remova código inútil e reorganize o programa para torná-lo tão fácil de ler quanto possível. Por exemplo, se você suspeita que o problema está numa parte profundamente aninhada do programa, tente reescrever aquela parte com uma estrutura mais simples. Se você suspeita de uma função longa, tente dividi-la em funções menores e testá-las separadamente.

Muitas vezes o processo de encontrar o caso de teste mínimo leva você ao erro. Se você descobrir que o programa funciona em uma situação, mas não em outra, você já tem uma boa dica a respeito do que está acontecendo.

De forma semelhante, reescrever pedaços de código pode ajuda a encontrar erros sutis. Se você faz uma alteração que você pensa que não afeta o programa, e ela afeta, você também tem uma dica.

23.3 A.3 Erros de semântica

De certa forma, os erros de semântica são os mais difíceis de depurar, porque o compilador e o sistema de tempo de execução não fornecem informações sobre o que está errado. Somente você sabe o que o programa deve fazer, e somente você sabe que ele não está fazendo isto.

O primeiro passo é fazer uma conexão entre o texto do programa e o comportamento que você está vendo. Você precisa de uma hipótese sobre o que o programa está realmente fazendo. Uma das coisas que dificultam é que os computadores trabalham muito rápido.

Você sempre desejará que a velocidade do programa pudesse ser diminuída para a velocidade humana, e com alguns depuradores você pode. Mas o tempo que leva para inserir umas poucas declarações `print` bem colocadas é geralmente mais curto quando comparado a configurar um depurador, inserir e remover *breakpoints*, e "caminhar" pelo programa até onde o erro ocorre.

23.3.1 A.3.1 Meu programa não funciona

Você deveria se fazer as seguintes perguntas:

- Há alguma coisa que o programa deveria fazer, mas que não parece que estar acontecendo?

- Não está acontecendo alguma coisa que não deveria acontecer? Encontre o código no seu programa que executa a função e veja se ele está executando no momento errado ou de forma errada.
- Uma parte do código está produzindo o efeito esperado? Cetifique-se que você entende o código em questão, especialmente se ele envolve chamadas de funções ou métodos em outros módulos da linguagem. Leia a documentação das funções e módulos que você está utilizando. Teste as funções escrevendo casos simples de teste e verificando os resultados.

Para programar, você precisa ter um modelo mental de como seus programas trabalham. Se você escreve um programa que não faz aquilo que você espera, é muito comum que o problema não esteja no programa, mas sim no seu modelo mental.

A melhor maneira de corrigir seu modelo mental é quebrar o programa em seus componentes (geralmente as funções e métodos) e testar cada componente de forma independente. Uma vez que você tenha encontrado a diferença entre seu modelo e a realidade, você pode resolver o problema.

Obviamente, componentes devem ser desenvolvidos e testado à medida que o seu programa vai ganhando vida. Se você encontra um problema, haverá uma pequena quantidade de novo código com funcionamento incerto.

23.3.2 A.3.2 Eu tenho uma grande expressão cabeluda e ela não faz o que eu espero

Escrever expressões complexas é legal se elas forem legíveis, mas pode ser difícil de depurar. Geralmente é uma boa ideia quebrar uma expressão complexa em uma série de atribuições para variáveis temporárias.

Por exemplo:

```
>>> self.mao[i].adicionaCarta(self.mao[self.encontraVizinho(i)].popCarta())
```

Isto pode ser escrito como:

```
>>> vizinho = self.encontraVizinho(i)
>>> cartaPega = self.mao[vizinho].popCarta()
>>> self.mao[i].adicionaCarta(cartaPega)
```

A versão explícita é mais fácil de ler, pois os nomes das variáveis fornecem documentação adicional, e é mais fácil depurar, já que você pode verificar os tipos das variáveis intermediárias e mostrar seus valores.

Um outro problema que pode ocorrer com expressões longas é que a ordem de avaliação pode não ser o que você espera. Por exemplo, se você está traduzido a expressão *x / 2pi* (XXX fazer a equação matemática) para Python, você poderia escrever:

```
>>> y = x / 2 * math.pi
```

Isto está correto por que a multiplicação e a divisão possuem a mesma precedência e são avalidadas da esquerda pra direita. Então a expressão calcula **x*pi/2* (XXX fazer a equação matemática).

Uma boa maneira de depurar expressões é adicionar parênteses para tornar a ordem de avaliação explícita:

```
>>> y = x / (2 * math.pi)
```

Sempre que você não estiver seguro sobe a ordem de avaliação, utilize parênteses. Não apenas o programa estará correto(no sentido de fazer aquilo que você tinha a intenção), ele também será mais legível por outras pessoas que não tenham memorizado as regras de precedência.

23.3.3 A.3.3 Eu tenho uma função ou método que não retorna o que eu espero

Se você possui uma declaração *return* com uma expressão complexa, você não tem a chance de exibir o valor de *return* antes que ele seja devolvido. Novamente, você pode utilizar uma variavel temporária. Por exemplo, ao invés de:

```
>>> def pega_encontrados(self):
    return self.mao[i].devolveEncontrados()
```

você poderia escrever:

```
>>>     def pega_encontrados(self):
    qtEncontrados = self.mao[i].devolveEncontrados()
    return qtEncontrados
```

Agora você tem a oportunidade de mostrar o valor de *qtEncontrados* antes de devolvê-lo.

23.3.4 A.3.4 Eu estou empacado mesmo e eu preciso de ajuda

Primeiro,tente sair da frente do computador por alguns minutos. Computadores emitem ondas que afetam o cérebro, causando estes efeitos:

- Frustação e/ou raiva.
- Crenças superticiosas ("o computador me odeia") e pensamentos mágicos ("o prorama só funciona quando eu coloco meu chapéu de trás pra frente").
- Programação pelo caminhar aleatório (a tentativa de programar escrevendo cada programa possível e escolhendo aquele que faz a coisa certa).

Se você estiver sofrendo de qualquer um destes sintomas, levante-se e vá dar uma caminhada. Quando você estiver calmo, pense no programa. O que ele está fazendo? Quais são as possíveis causas do comportamento inadequado? Quando foi a última vez que você teve um programa funcionando, e o que você fez depois disto?

Às vezes é uma questão de tempo para encontrar um erro. Nós geralmente encontramos os erros quando estamos longe do computador e deixamos nossa mente vaguear. Alguns dos melhores lugares para encontrar erros são trens, chuveiros e na cama, logo antes de pegar no sono.

23.3.5 A.3.5 Não, eu preciso mesmo de ajuda

Isto acontece. Mesmo os melhores programadores empacam de vez em quando. Às vezes você trabalha num programa por tanto tempo que não consegue ver o erro. Um par fresco de olhos é o que se precisa.

Antes de trazer mais alguém, certifique-se de que você tenha esgotado as técnicas descritas aqui. Seu programa deve ser tão simples quanto possível, e você deve estar trabalhando com a mais simples das entradas que causam o erro. Você deve ter declarações *print* nos lugares apropriados (sem comprometer a compreensividade da saída do programa). Você tem que entender o problema o suficiente para descrevê-lo concisamente.

Quando você trouxer alguém pra ajudar, assegure-se de dar a este alguém a informação que ele precisa:

- Se existe uma mensagem de erro, o que é ela e que parte do programa ela indica?
- Qual foi a última coisa que você fez antes deste erro acontecer? Quais foram as últimas linhas de código que você escreveu, ou qual é o novo caso de teste que falha?
- O que você já tentou até o momento, e o que você aprendeu?

Quando você encontrar um erro, gaste um segundo para pensar sobre o que você poderia fazer para encontrá-lo mais rápido. Da próxima vez que você ver algo similar, você terá condições de encontrar o erro mais rapidamente.

Lembre-se, o objetivo não é apenas fazer o programa funcionar. O objetivo é aprender como fazer o programa funcionar.

CAPÍTULO 24

Apêndice B: Criando um novo tipo de dado

Tópicos

Em linguagens com suporte à orientação a objetos, programadores podem criar novos tipos de dados que se comportam de forma semelhante aos tipos de dados built-in. Vamos explorar esse recurso criando uma classe chamada `Fracao`. Esta classe terá comportamento muito semelhante aos tipos numéricos da linguagem: `int`, `long` e `float`.

Frações, também conhecidas como números racionais, são valores que podem ser expressos como uma razão de dois números inteiros, por exemplo, 5/6. No exemplo fornecido, o 5 representa o numerador, o número que fica em cima, que é dividido, e o 6 representa o denominador, o número que fica embaixo, pelo qual a divisão é feita. A fração 5/6 pode ser lida "cinco dividido por seis".

O primeiro passo é definir a classe `Fracao` com o método `__init__` que recebe como parâmetros o numerador e o denominador, sendo ambos do tipo `int`:

```
class Fracao:
    def __init__(self, numerador, denominador=1):
        self.numerador = numerador
        self.denominador = denominador
```

A passagem do denominador é opcional. Uma Fracao com apenas um parâmetro representa um número inteiro. Sendo o numerador n, a fração construída será n/1.

O próximo passo é escrever o método `__str__` que exibe as frações corretamente: a forma numerador/denominador.

```
class Fracao:
    ...
```

```
    def __str__(self):
        return "%d/%d" %(self.numerador, self.denominador)
```

Para testar o que foi feito até aqui, salve a classe `Fracao` em um arquivo chamado `Fracao.py` e importe-a no interpretador interativo. Criaremos uma instância da classe e imprimiremos ele na tela:

```
>>> from Fracao import Fracao
>>> numero = Fracao(5, 6)
>>> print "A fração é ", numero
A fração é 5/6
```

Como de costume, o comando `print` chama implicitamente o método `__str__`.

24.1 B.1 Multiplicação de frações

É interessante que nossas frações possam ser somadas, subtraídas, multiplicadas, divididas, etc. Enfim, todas as operações matemáticas das frações. Para que isso seja possível, vamos usar o recurso de sobrecarga de operadores.

Começaremos pela multiplicação por que é a operação mais fácil de ser implementada. Para multiplicar duas frações, criamos uma nova fração, onde o numerador é o produto dos numeradores das frações multiplicadas e o denominador é o produto dos numeradores das frações multiplicadas. O método utilizado em Python para sobrecarga do operador `*` chama-se `__mul__`:

```
class Fracao:
    ...
    def __mul__(self, other):
        return Fracao(self.numerador * other.numerador,
                      self.denominador * ohter.denominador)
```

Vamos testar este método criando e multiplicando duas frações:

```
>>> print Fracao(5, 6) * Fracao(3, 4)
15/24
```

O método funciona, mas pode ser aprimorado! Podemos melhorar o método visando possibilitar a multiplicação de uma fração por um inteiro. Usaremos a função `isinstance` para verificar se o objeto `other` é um inteiro, para em seguida convertê-lo em uma fração.

```
class Fracao:
    ...
    def __mul__(self, other):
        if isinstance(other, int):
            other = Fracao(other)
        return Fracao(self.numerador * other.numerador,
                      self.denominador * ohter.denominador)
```

Agora conseguimos multiplicar funções por inteiros, mas só se a fração estiver à esquerda do operador `*`. Vejamos um exemplo em que nossa multiplicação funciona e outro no qual ela não funciona:

```
>>> print Fracao(5, 6) * 4
20/6
>>> print 4 * Fracao(5, 6)
TypeError: __mul__ nor __rmul__ defined for these operands
```

O erro nos da uma dica: não mexemos em nenhum `__rmul__`.

Para realizar a multiplicação, busca no elemento à esquerda do operador `*` o método `__mul__` compatível com a multiplicação realizada (no nosso caso, que receba um inteiro e uma fração, nesta ordem). Se o método não for encon-

trado, o Python buscará no elemento à direita do operador `*` o método `__rmul__` compatível com a multiplicação realizada (que então deve ser lida da direita para a esquerda). Como a multiplicação é lida da direita para a esquerda, temos que o nosso método `__rmul__` deve ser igual ao método `__mul__` implementado anteriormente.:

```
class Fracao:
    ...
    __rmul__ = __mul__
```

Fazendo assim, dizemos que o método `__rmul__` funciona da mesma forma que o método `__mul__`. Agora, quando multiplicarmos `4 * Fracao(5, 6)`, o interpretador Python chamará o método `__rmul__` do objeto `Fracao`, passando o 4 como parâmetro.

```
>>> print 4 * Fracao(5, 6)
20/6
```

Como o método `__rmul__` é também o método `__mul__`, e o método `__mul__` consegue trabalhar com parâmetro do tipo inteiro, nossa multiplicação está completa.

24.2 B.2 Soma de frações

Somar é mais complicado do que multiplicar, pelo menos quando estamos somando frações e temos que implementar isso em uma linguagem de programação. Mas não se assuste, não é tão complicado assim. A soma de a/b com c/d é (a*d+c*b)/(b*d).

Tomando a multiplicação como base, podemos facilmente implementar os métodos `__add__` e `__radd__`:

```
class Fracao:
    ...
    def __add__(self, other):
        if isinstance(other, int):
            other = Fracao(other)
        return Fracao(self.numerador   * other.denominador +
                      self.denominador * other.numerador,
                      self.denominador * other.denominador)

    __radd__ = __add__
```

Podemos testar o método usando frações e inteiros:

```
>>> print Fracao(5, 6) + Fracao(5, 6)
60/36
>>> print Fracao(5, 6) + 3
23/6
>>> print 2 + Fracao(5, 6)
17/6
```

Os dois primeiros exemplos chamam o método `__add__`, enquanto o último exemplo chama o método `__radd__`.

24.3 B.3 Simplificando frações: O algoritmo de Euclides

No exemplo anterior, calculamos a soma de 5/6 com 5/6 e obtivemos o resultado 60/36. O resultado está correto, porém não está representado na melhor forma possível. O ideal é simplificarmos a fração. Para simplificar ao máximo esta fração, devemos dividir o numerador e o denominador pelo máximo divisor comum (MDC) deles, que é 12. Fazendo isso, chegamos à forma mais simples da fração, que é 5/3.

De forma geral, sempre que um objeto do tipo `Fracao` for criado, a fração deve ser simplificada, através da divisão do numerador e do denominador pelo seu MDC. Quando a fração já está em sua forma mais simples, o MDC vale 1.

Euclides de Alexandria (aprox. 325 a. C. – 365 a. C.) desenvolveu um algoritmo para encontrar o MDC de dois inteiros `m` e `n`:

> Se `n` é múltiplo de `m`, então `n` é o MDC. Caso contrário, o MDC é o MDC de `n` e o resto da divisão de `m` por `n`.

Esta definição recursiva pode ser representada de forma concisa pela função:

```
def mdc(m, n):
    if m % n == 0:
        return n
    else:
        return mdc(n, m % n)
```

Na primeira linha da função, utilizamos o operador de módulo para checar se `m` é divisível por `n`. Na última linha, usamos o mesmo operador para obter o resto da divisão de `m` por `n`.

Já que todas as operações que escrevemos criam um novo objeto do tipo `Fracao`, podemos utilizar a função `mdc` no método `__init__` de nossa classe:

```
class Fracao:
    def __init__(self, numerador, denominador=1):
        m = mdc(numerador, denominador)
        self.numerador = numerador / m
        self.denominador = denominador / m
```

Agora, sempre que criarmos uma fração, ela será reduzida:

```
>>> Fracao(100, -36)
-25/9
```

Sempre que o denominador é negativo, o sinal negativo deve passar para o numerador. O interessante é que, ao usarmos o Algoritmo de Euclides, esta operação ocorre de forma transparente.

Antes de seguirmos para o próximo passo, vamos ver como está nossa classe `Fracao` completa?

```
class Fracao:
    def __init__(self, numerador, denominador=1):
        m = mdc(numerador, denominador)
        self.numerador = numerador / m
        self.denominador = denominador / m

    def __str__(self):
        return "%d/%d" %(self.numerador, self.denominador)

    def __mul__(self, other):
        if isinstance(other, int):
            other = Fracao(other)
        return Fracao(self.numerador * other.numerador,
                      self.denominador * ohter.denominador)

    __rmul__ = __mul__

    def __add__(self, other):
        if isinstance(other, int):
            other = Fracao(other)
        return Fracao(self.numerador   * other.denominador +
                      self.denominador * other.numerador,
```

```
                    self.denominador * other.denominador)

    __radd__ = __add__
```

24.4 B.4 Comparando frações

Suponha que tenhamos duas frações (instâncias da classe `Fracao`), `a` e `b`, e nós fazemos a comparação `a == b`. A implementação padrão do operador == realiza um "teste raso", ou seja, verifica se `a` e `b` são o mesmo objeto.

Queremos personalizar este retorno, de forma que a comparação retorne `True` se `a` e `b` tiverem o mesmo valor, o que é chamado de "teste profundo".

Temos que ensinar às frações como elas devem se comparar. Como foi visto na seção 15.4, todos os operadores de comparação podem ser sobrecarregados por apenas um método: `__cmp__`.

Por convenção, o método `__cmp__` retorna um número negativo se `self` for menor que `other`, zero se eles forem iguais e um número positivo se `self` for maior que `other`.

A forma mais simples de comparar frações é através da multiplicação cruzada (denominador * numerador e vice-versa). Se `a/b > c/d`, então `ad > bc`. Tendo isso em mente, vamos criar o `__cmp__`:

```
class Fracao:
    ...
    def __cmp__(self, other):
        diferenca = (self.numerador * other.denominador -
                     self.denominador * other.numerador)
        return diferenca
```

Se `self` for maior que `other`, a `diferenca` será positiva. Se `other` for maior, a `diferenca` será negativa. Se os dois forem iguais, a `diferenca` será zero.

24.5 B.5 Indo mais além...

Obviamente, não terminamos de representar uma fração. Ainda temos que implementar a subtração sobrescrevendo o método `__sub__` e a divisão sobrescrevendo o método `__div__`.

Uma forma de tratar tais operações é sobrescrever os métodos de negação (`__neg__`) e de inversão (`__invert__`). Assim, podemos fazer a subtração através da negação do elemento à direita do operador e somando, e podemos fazer a divisão através da inversão do elemento à direita do operador e multiplicando.

Depois, temos que implementar os métodos `__rsub__` e `__rdiv__`. Infelizmente, não podemos utilizar o mesmo "macete" utilizado para multiplicação e soma, por que a divisão e a subtração não são comutativas, ou seja, a ordem dos fatores altera o resultado final.

As negações unárias, que representam o uso do sinal de negação com apenas um elemento, são implementadas através da sobrescrita do método `__neg__`.

Potências podem ser calculadas através do método `__pow__`, mas a implementação é um pouco complicada. Se o expoente da potência não for um inteiro, o resultado provavelmente não poderá ser representado como uma fração. Por exemplo, `Fracao(2) ** Fracao(2)` é a raiz quadrada de 2, que é um número irracional, e números irracionais não pode ser representados por frações. Logo, é uma tarefa complicada implementar uma versão genérica do método `__pow__`.

Existe, ainda, uma outra extensão para a classe `Fracao` que pode vir à mente. Até aqui, assumimos que o numerador e o denominador são números inteiros.

24.5.1 B.5.1 Exercício

Como exercício, finalize a implementação da classe `Fracao`, tornando-a capaz de tratar subtração, divisão e potenciação.

24.6 B.6 Glossário

máximo divisor comum (MDC) O maior inteiro positivo que tem como múltiplo o numerador e o denominador de uma fração.

negação unária É a operação que calcula a inversão de um elemento, usualmente representada com um sinal de menos − à esquerda do elemento. É chamada unária pelo contraste com a operação binária que usa o sinal de menos, a subtração.

simplificar Transformar uma fração em sua equivalente com o MDC valendo 1

CAPÍTULO 25

Apêndice C: Leituras recomendadas

Tópicos

25.1 C.1 Recomendações para leitura

E agora, para onde você vai? Existem diversas direções que podem ser seguidas, aumentando o seu conhecimento especificamente em Python e/ou na ciência da computação em geral.

Os exemplos contidos neste livro foram deliberadamente simples, ou seja, eles podem não ter mostrado todo o potencial do Python. Abaixo uma lista de extensões e sugestões para projetos que usem Python:

- Programar utilizando interfaces gráficas (GUI - graphical user interface) permite ao seu programa usar um ambiente de janelas para interagir com o usuário e exibir conteúdos gráficos (janelas, imagens, etc.) A biblioteca gráfica mais antiga para Python é o Tkinter, que é baseado no Tcl criado por Jon Ousterhout e na linguagem de script Tk. O Tkinter está incluso na distribuição padrão do Python, ou seja, quando instalamos o Python, o módulo do Tkinter é instalado também.

  Outra biblioteca famosa é o wxPython, que é essencialmente uma "folheagem" do Python sobre o wxWindows, uma biblioteca C++ que, por sua vez, implementa janelas usando a interfaces nativas nas plataformas Windows e Unix (incluindo Linux). As janelas e controles no wxPython tendem a ter um visual mais nativo do que no Tkinter e são mais um pouco mais simples de usar.

  Em qualquer biblioteca GUI que você usar, você usará a programação orientada a eventos, onde o usuário, e não o programador, determina o fluxo de execução. Este estilo de programação exige um novo costume que, por vezes, o obrigará a repensar toda a estrutura de um programa.

- A programação web é um modelo de programação que integra o Python com a Internet. Por exemplo, você pode criar um cliente web que abre e lê uma página remota (quase) tão facilmente como você pode abrir um arquivo local. Existem, ainda, módulos para Python que permitem acesso remoto a arquivos utilizando FTP, e módulos que permitem você enviar e receber e-mails. Python também é usado (amplamente) para construir programas em servidores web com o intuito de tratar dados fornecidos por formulários.

- Banco de dados são como super arquivos, que armazenam dados em esquemas predefinidos, e mantém relações entre itens de dados que lhe permitem acessar os dados de várias formas. Python tem vários módulos que possibilitam ao usuário conectar seus programas a diversos sistemas gerenciadores de banco de dados, tanto sistemas livres (Open source) quanto sistemas comerciais.
- A programação multitarefa (multithread) permite que você execute várias tarefas (threads) dentro de um único programa. Se você já teve a experiência de usar um navegador web para se deslocar por uma página web enquanto o navegador ainda está carregando ela, então você já tem uma ideia do que da pra fazer usando a programação multitarefa.
- Quando o desempenho é primordial, você pode escrever extensões para o Python em linguagens compiladas, como C e C++. Este abordagem é vastamente utilizada na biblioteca padrão do Python, formando a sua base. O mecanismo de ligação de dados e funções é um pouco complexo. Existe uma ferramenta, chamada SWIG (Simplified Wrapper and Interface Generator), que faz este processo de ligação ser mais simples.

25.2 C.2 Sites e livros sobre Python

Aqui estão algumas recomendações do autor de informações sobre Python na Internet:

- A página oficial do Python (www.python.org) é o ponto de partida para pesquisa sobre qualquer material ligado a Python. Lá você encontrará ajuda, documentação, links para outros sites e listas de discussão nas quais você pode participar.
- O Open Book Project (www.ibiblio.com/obp) contém não apenas este livro, mas também livros similares que abordam Java e C++, escritos por Allen Downey. Além disso, há aulas sobre Circuitos Elétricos feitas por Tony R. Kuphaldt; "Get down with ...", um conjunto de tutoriais sobre uma gama de tópicos em ciência da computação, escrito e editado por alunos de ensino médio; "Python for Fun", um conjunto de estudos de caso em Python, feito por Chris Meyers; e "The Linux Cookbook", escrito por Michael Stultz, com 300 páginas de dicas e técnicas.
- Por último, se você for ao Google e buscar por "python -snake -monty", você encontrará cerca de 750 mil resultados.

E aqui estão alguns livros que contém material sobre Python:

- Core Python Programming, escrito por Wesley Chun, é um grande livro com cerca de 750 páginas. A primeira párte do livro apresenta os recursos básicos do Python. A segunda parte traz uma introdução aos tópicos mais avançados, incluindo muitos dos mencionados acima.
- Python Essential Reference, escrito por David M. Beazley, é um livro pequeno, mas que contém informações tanto da linguagem em si quanto dos módulos da biblioteca padrão. É também muito bem indexado.
- Python Pocket Reference, escrito por Mark Lutz, este livro realmente cabe no seu bolso. Embora não seja tão abrangente quanto o "Python Essential Reference", "Python Pocket Reference" é uma referência para se ter em mãos o tempo todo, capaz de atender muito bem à explicação das funções e métodos mais comuns. Mark Luiz também é autor do livro "Programming Python", um dos primeiros (e maiores) livros sobre Python, e não visa o programador iniciante. Seu último livro, "Learning Python", é menor e mais acessível.
- Python Programming on Win32, escrito por Mark Hammond e Andy Robinson, é um "tem que ter" para qualquer pessoa utilizando Python seriamente para desenvolver aplicações Windows. Entre outras coisas, o livro apresenta a integração entre Python e COM, cria uma pequena aplicação com wxPython, e ainda usar Python para criar scripts para aplicações como Word e Excel.

25.3 C.3 Livros de ciência da computação recomendados

As seguintes sugestões de leitura incluem muitos dos livros favoritos do autor. Eles lidam com as boas práticas de programação e ciência da computação em geral.

- The Practice of Programming, escrito por Kernighan e Pike, abrange não apenas o projeto e a implementação de algoritmos e estrutura de dados, mas também depura, testa e melhora o desempenho de programas. Os exemplos são principalmente em C++ e Java, nenhum em Python.
- The Elements of Java Style, editado por Al Vermeulen, é outro livro pequeno que discute alguns dos mais finos pontos de boas práticas de programação, como o bom uso de conveções, comentários, e ainda espaços em branco e endentação (o que não é um problema em Python). O livro também abrange programação por contrato, usando asserções para capturar erros testando precondições e poscondições, e programação adequada utilizando threads e sua sincronização.
- Programming Pearls, escrito por Jon Bentley, é um livro clássico. O livro consiste de casos de estudo que originalmente apareceram na coluna do autor no site Communications of ACM (Association for Computing Machinery). Os estudos lidam com trade-offs em programação e por que isto é, muitas vezes, uma péssima idéia, especialmente para usar na sua primeira ideia para um programa. O livro é um pouco mais velho que os outros acima (1986), então os exemplos estão em linguagens antigas. Existem vários problemas para resolver, uns com solução e outros com dicas. O livro foi muito famoso e foi seguido por um segundo volume.
- The New Turing Omnibus, escrito por A.K Dewdney, fornece uma leve introdução a 66 tópicos de ciência da computação, indo de computação paralela aos vírus de computador, de tomografias computadorizadas a algoritmos genéticos. Todos os tópicos são curtos e agradáveis. Um livro anterior escrito por Dewdney, The Armchair Universe, é uma coleção de sua coluna "Computer Recreations" (Brincadeiras computacionais) na revista Scientific American. Ambos os livros representam uma rica fonte de ideias para projetos.
- Turtles, Termites and Traffic Jams, escrito por Mitchel Resnick, trata do poder de descentralização e como um comportamento complexo pode ocorrer a partir de simples atividades coordenadas, com um grande número de agentes. A execução do programa demonstra o comportamento complexo, que é, muitas vezes, contraintuitivo.
- Gödel, Escher and Bach, escrito por Douglas Hofstadter. Simplificando, se você encontrar a magia na recursão, você vai encontrar também neste best-seller. Um dos temas abordados por Hofstadter envolve "loops estranhos" onde os padrões evoluem e ascendem até se encontrarem novamente. Esta é a controvérsia de Hofstadter, de que tais "loops estranhos" representam o elemento essencial que separa o animado do inanimado. Ele demonstra tais padrões na música de Bach, nos quadros de Escher e na incompletude dos teoremas de Gödel.

CAPÍTULO 26

Apêndice D: GNU Free Documentation License

Coligi aqui os links para traduções da Licensa Pública GNU fornecidos pelo professor Imre Simon:

“http://creativecommons.org/licenses/GPL/2.0/legalcode.pt“

“http://www.magnux.org/doc/GPL-pt_BR.txt“

“http://www.ead.unicamp.br/minicurso/bw/texto/fdl.pt.html” (esta não consegui abrir...)

Encontrei também uma reprodução neste texto em pdf, aliás sobre o incentivo do governo ao uso do **Software Livre**:

“http://www.inf.ufpr.br/info/techrep/RT_DINF004_2002.pdf.gz“

Parecem haver pequenas discrepâncias entre as várias traduções, mas acho que deveríamos escolher uma para publicar aqui como referência.

CAPÍTULO 27

Indices and tables

- *genindex*
- *modindex*
- *search*